Performance Analysis of
Multi-Core Multi-Mode Systems with Shared Resources
- Principles and Application to AUTOSAR -

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors

der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von Mircea Florin Negrean

aus Carei

eingereicht am 15.06.2015

mündliche Prüfung am 21.08.2015

1. Referent: Prof. Dr.-Ing. Rolf Ernst
2. Referent: Associate Prof. Dr.-Ing. Paul Pop
3. Referent: Prof. Dr.-Ing. Harald Michalik (Vorsitzender)

Druckjahr: 2016

Dissertation an der Technischen Universität Braunschweig,
Fakultät für Elektrotechnik, Informationstechnik, Physik

Abstract

Embedded systems, as a union of computing hardware and software, are integrated in many electric and electronic devices in order to implement diverse functions which allow an enhancement of the human life with respect to safety, security, comfort, autonomy and productivity. Many of these functions have not only to produce correct results but also to supply them in a time bounded interval. Software applications which implement functions with stringent timing requirements are called real-time applications and embedded systems hosting them are called real-time systems.

Nowadays, the number and complexity of timing critical functions used in various application domains is steadily increasing. A key example is the automotive domain, one of the main technology drivers worldwide, where more and more functions are implemented in powertrain systems, advanced driver assistance systems or infotainment systems in order to reduce pollution, to increase the safety on the roads or to enhance the driving comfort. The number and the computational complexity of such functions demand for more computational resources. In order to satisfy the rising computational demands, embedded systems are turning to multi-core architectures. However, while multi-core solutions generally deliver additional performance more cost efficiently, their applicability is challenged by the additional execution delays that tasks will experience due to contention on shared resources (e.g. shared memories or semaphores). In this context, the development process of multi-core real-time systems imply a careful investigation of their timing behavior which requires appropriate methods and tools for timing and performance verification.

Previous work from academia and industry showed that formal performance analysis approaches are well suited for the analysis of multiprocessor and distributed real-time systems. However, the applicability of the current methodology is still limited as many system details are not covered on the modeling and analysis side. This thesis provides new analysis methods which extend the scope of formal performance analysis and thus enable the investigation of new design options for real-time systems. The main contributions of the present thesis can be summarized as follows:

• First, this thesis proposes novel approaches for the analysis of worst-case blocking- and response-times for static (i.e. single-mode) real-time applications that share resources in partitioned multi-core systems, i.e. in multi-core setups were applications are statically mapped to the processor cores, which are then individually scheduled at run-time. For this purpose a compositional performance analysis methodology is adopted and extended to take into account the contention of tasks on the processor cores and on the shared resources. Unlike existing analysis methods, the solutions proposed in this thesis cover realistic system configurations with tasks that exhibit arbitrary activations and

deadlines and rely on a sophisticated model to capture the load imposed on the shared resources and the timing between individual requests for shared units. Furthermore, in comparison to previous work, the new analysis approaches are dedicated to partitioned multi-core setups in which not only preemptive but also non-preemptive scheduling can be combined with different shared resource arbitration strategies, proposed by academia and industry. All these extend the applicability of formal performance analysis to industry specific setups, as for example for the current generation of automotive AUTOSAR conform multi-core controllers where preemptive and non-preemptive scheduling can coexist on each processor core and arbitrarily activated tasks can share common resources (e.g. "lock" protected semaphores) according to a spinlock-based synchronization policy.

The new analysis methods are applied for investigating the impact of different design decisions regarding task scheduling and shared resource arbitration on the timing behavior of multi-core real-time applications.

• Further, this thesis proposes novel timing analysis solutions for multi-mode real-time systems, i.e. for systems which adapt their behavior during runtime to changing conditions in the environment, switch to an emergency state or change their resource usage. The adaptive nature of these systems imply a complex timing behavior, characterized by dynamic changes of the timing properties at runtime, which is difficult to capture by formal analysis methods.

For such systems, the settling time of a mode change, called mode change transition latency, is identified as an important system parameter that has been neglected before. Known approaches that address the problem of timing analysis for multi-mode real-time systems are restricted to applications without communicating tasks. Also, these assume that transitions between operational modes are initiated only during a steady state, however, without indicating when a system executes in a steady state. This thesis contributes a novel analysis algorithm which gives a maximum bound on each mode change transition latency of multi-mode distributed applications.

• Finally, this thesis addresses the problem of designing and analyzing multi-mode applications, which share resources on multi-core systems, in the context of the automotive AUTOSAR specifications. For this purpose, an approach for safely handling shared resources across mode changes is discussed and a corresponding timing analysis method is developed. The new analysis solution combines modeling and analysis elements of the multi-core and multi-mode related analysis solutions. This enables system designers to handle the timing behavior of more complex systems in which the problems of mode management, multi-core scheduling and shared resource arbitration coexist. The new analysis methods proposed for multi-mode real-time systems rely on and extend the compositional performance analysis methodology adopted before for the analysis of static multi-core applications. Their applicability is demonstrated by experimental data and emphasized by an automotive specific use case.

To summarize, the contribution of this thesis is a comprehensive performance analysis framework for static and multi-mode real-time applications which share resources on multi-core systems.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Rolf Ernst for giving me the opportunity to work in a highly professional and dynamic academic environment. Thank you for sharing your profound knowledge with me, for your support and valuable guidance and for the possibility to combine scientific research with industry projects.

I would also like to thank Prof. Paul Pop for kindly agreeing to be the co-examiner of this thesis and for his helpful feedback.

I am thankful to all the members of the administrative and technical staff of the Institute for Computer and Network Engineering (IDA) at the TU Braunschweig for their prompt and valuable support as well as for their hospitality and friendship.

My sincere thanks go to all the IDA-colleagues and students for provinding a nice work atmosphere during the years we shared working together on various projects, for the many interesting and productive discussions as well as for the good time during conferences and after work. I am happy that many of you became my friends and that with some of you I have the possibility to work further on.

I also thank the fellow researchers and professors from ETH Zürich, Linköping University, Technical University of Denmark and University of Porto for their valuable contributions and the great time in joint research projects.

I am also thankful to many people from Romania, professors, neighbors, friends and family members, who guided, supported and encouradged me during school, studies and in everything that means life. I remember you all.

Most importantly, I am deeply grateful to my parents Marioara and Traian and to my grandparents for their unconditional love, patience and the confidence in me and in my decisions. Without your constant support I would not be the person that I am today and this work would not have been possible.

Finally, I am profoundly grateful to my wife Roxana. Thank you for giving me strength and for encouraging me during all these years. Thank you for your endless love and patience.

Contents

1 Introduction **11**

1.1 Embedded Systems: Multi-Core Architectures and Corresponding Design Challenges . . . 14

1.2 Handling Timing Aspects in the Development Process of Embedded Real-Time Systems . . . 20

1.2.1 Timing-Aware Development Process . . . 20

1.2.2 Current Practice: Solutions to Handle and Verify the Timing Behavior . . . 23

1.3 Thesis Contribution and Outline . . . 25

2 System Modeling and System-Level Performance Analysis **29**

2.1 Survey on System-Level Performance Analysis Approaches . . . 29

2.2 System Model . . . 31

2.2.1 General System Model . . . 32

2.2.2 Timing Model . . . 33

2.2.3 System Model and Task State Model: Example . . . 36

2.3 Compositional System-Level Performance Analysis Procedure for Multi-Core Systems with Shared Resources . . . 39

2.3.1 General Analysis Procedure . . . 39

2.3.2 Solving the System-Level Iterative Analysis Procedure . . . 42

2.4 Summary and Overview . . . 45

3 Timing Analysis of Multi-Core Systems with Shared Resources **47**

3.1 Introduction . . . 47

3.2 Related Work . . . 49

3.2.1 Multiprocessor Scheduling . . . 49

3.2.2 Resource Sharing in Multiprocessor Systems . . . 55

3.3 Multi-Core System Model . . . 59

3.4 Impact of Multi-Core Design Decisions . . . 62

3.5 Principle of the Response-Time Analysis Procedures for Multi-Core Systems with Shared Resources . . . 69

3.5.1 Response Time Analysis of Arbitrarily Activated Tasks in Single-Core Processor Systems . . . 70

3.5.2 Extending Uniprocessor Scheduling Theory . . . 74

3.6 Derivation of the Shared Resource Load . . . 74

3.7 Response-Time Analysis for Partitioned Static Priority Preemptive Scheduling in Multi-Core Systems with Shared Resources . . . 78
3.7.1 Blocking Time Analysis for MPCP . . . 78
3.7.2 Response Time Analysis for Partitioned Multi-Core SPP Scheduling 81
3.8 Response-Time Analysis for Partitioned Static Priority Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources . . . 83
3.8.1 Blocking Time Analysis for Multi-Core SPNP Scheduling . . . 84
3.8.2 Response Time Analysis for Partitioned Multi-Core SPNP Scheduling . . . 87
3.9 Response-Time Analysis for AUTOSAR conform Multi-Core ECUs . . . 91
3.9.1 Extended Multi-Core System and Scheduling Model . . . 91
3.9.2 Blocking Time Analysis for AUTOSAR conform Multi-Core ECUs 95
3.9.3 Response Time Analysis for Partitioned AUTOSAR Scheduling . . 103
3.10 System-Level Analysis Integration . . . 110
3.11 Experimental evaluation . . . 114
3.11.1 Evaluation of Multi-Core Setups under Partitioned SPP Scheduling and MPCP Shared Resource Arbitration . . . 115
3.11.2 Evaluation of Multi-Core Setups under Partitioned SPNP Scheduling and MLP-NP Shared Resource Arbitration . . . 118
3.11.3 Evaluation of AUTOSAR conform Multi-Core Setups . . . 121
3.12 Summary . . . 128

4 Timing Analysis of Multi-Mode Applications on Multi-Core Systems 129
4.1 Introduction . . . 129
4.2 Related Work . . . 131
4.3 System and Mode Change Model . . . 135
4.4 Bounding Mode Change Transition Latencies for Multi-Mode Real-Time Distributed Applications . . . 138
4.4.1 The Mode Change Recurrent Effect: Problem Statement and Analysis Concepts . . . 138
4.4.2 Analysis of Mode Change Transition Latencies . . . 141
4.4.3 Experiments . . . 152
4.4.4 Case Study . . . 154
4.5 Response-Time Analysis for Multi-Mode Applications on Multi-Core Systems with Shared Resources . . . 161
4.5.1 Multi-Mode Multi-Core System Model . . . 161
4.5.2 Handling Shared Resources in Multi-Mode Multi-Core Systems using AUTOSAR 4.0 . . . 162
4.5.3 Timing Analysis for Multi-Mode Multi-Core Systems with Shared Resources . . . 166
4.5.4 Experiments . . . 179
4.6 Summary . . . 181

5 Conclusion **183**
5.1 Future directions 185

6 List of publications **187**
6.1 With Relation to Thesis 187
6.2 Others 190

1 Introduction

During the last years the role of embedded systems significantly increased in all aspects of modern human life. Embedded systems, as part of electric and electronic (E/E) devices, can be found in many application domains such as consumer electronics, telecommunication, industrial and home automation, energy systems, transportation and medical equipments. Mobile phones, equipments for light and temperature control in household, flight guidance systems, adaptive cruise control (ACC) or electronic stability control systems (ESP) for vehicles, cardiac pacemakers and medical imaging equipments for the human body are only a few examples of devices employing embedded systems.

Embedded systems are essentially a union of computing hardware and software integrated into larger products and interfaced to the physical environment [85, 126]. In order to accomplish different domain specific purposes embedded systems implement complex functions that are performed by electronics and newly, more and more by software, as for example in the automotive domain [28]. Currently, automotive specific functions are implemented by multiple electronic control units (ECUs) which communicate and exchange sensor and actuator signals over dedicated field-buses and interconnects. Mid-2000s in a single high-end car there were almost 100 millions lines of code, between 70 and 100 distinct ECUs, more than 9 buses over which more than 6000 signals are transmitted [28, 123]. The steadily increasing demand for more features, safety, security, efficiency and not the last for lower cost triggers the implementation of new and often even more complex functions. An example from the automotive domain is given in Figure 1.1, which depicts the increasing number and complexity of E/E components in the Mercedes-Benz E-Class until the W212 model lauchend in 2009.

Figure 1.1: Growing complexity of E/E components and network communication (Source: Daimler AG Group Research and Advanced Engineering [155])

Advanced driver assistance systems comprising features like object recognition, night vision, lane-keeping, driver monitoring, car-to-x communication are nowadays developed with the goal of increasing the safety on the roads. Another example is the automotive power train domain where advanced engine control functions are implemented in order to maximize efficiency and reduce pollution to levels that fulfill the increasingly rigurous emission standards. Hundreds of other functions ranging from anti-lock braking systems to infotainment and internet based services target a safe and pleasant driver experience. As indicated by the results of a trend analysis until 2015 on automotive electronics market [163], illustrated in Figure 1.2, the amount and complexity of the automotive applications are only going to grow which result in an increased need for more computational power.

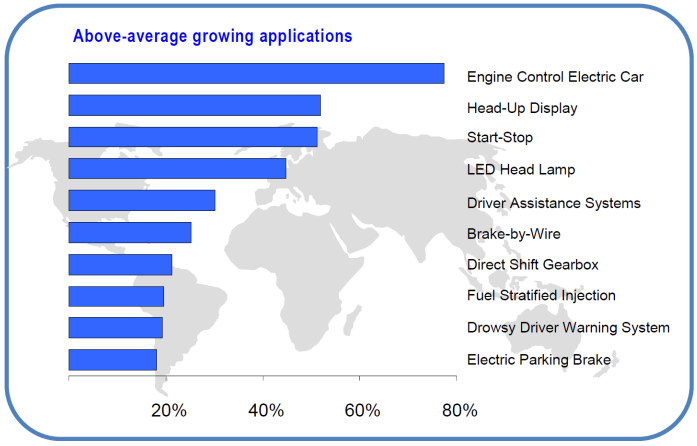


Figure 1.2: Top 10 above average automotive applications growth rates [163]

The classic approach followed for many years to achieve the required processor performance was to improve the level of function integration on a processor, e.g. by implementing multiple individual operational modes with different levels of resource usage, and to increase the processor operation frequency. However, in the context of the current rate of electronics' evolution the current embedded systems based on single-core processor architectures are approaching their performance limit [34]. Concerns regarding the electro-magnetic compatibility (EMC), the increased current consumption and the associated heat dissipation issues make the increase of the single-core processors' operating frequency infeasible.

In this context, embedded systems increasingly rely on multi-core architectures similar to servers and high-end computers many years ago. For example, dual-core processors just became state-of-the art in modern smartphones and tables but quad-core processors are already announced to join the new products [98, 113, 86]. Emerging medical imaging systems also take advantage of the multi-core processors capabilities and thus enable an increase of the quality of medical care while keeping the overall system costs affordable [112, 49, 51]. Strong control-dominated systems, as for example in automotive, also focus on multi-core architectures [34]. Freescale and Infineon, two of the main

worldwide semiconductor producers provide multi-core solutions for diverse automotive applications [50, 52, 66, 67]. This trend is confirmed not only by hardware solutions but also by standardization efforts on automotive software architectures [8]. At the end of 2009 specifications of the automotive standard AUTOSAR already included support for distributed execution of software on embedded multi-core processors [12].

However, the much awaited benefits with respect to energy consumption and processing power are not just simply coming with the increasing number of processing cores. The integration of multiple existing subsystems, e.g. single-core applications, in a multi-core system will not automatically lead to an exhaustive exploit of the available multi-core capabilities. It is well known that the multi-core hardware evolved and is still evolving faster than the dedicated software, which means that the available software was not developed for multi-core processors and obviously is not able to truly exploit multi-cores. Furthermore, resource contention in multi-core setups challenges the multi-core systems theoretical performance potential. Multiple applications mapped on distinct cores that are sharing the same system bus and memory units are strongly competing for the available bandwidth which inevitably reduces the expected speedup [125, 146, 124]. Even worse, uncoordinated accesses from different cores to the commonly shared resources may lead to unbounded blocking scenarios which endanger the correct systems' operation.

Correspondingly, a rapid paradigm shift from single-core to multi-core embedded solutions is accompanied by major challenges in system design and verification. Therefore, the evolution towards multi-core solutions has to be supported by a well developed design process that beside other aspects requires a rigorous understanding of the timing behavior in multi-core systems with shared resources. This is a key issue, because most of the complex features which will be implemented on multi-core systems rely on computationally intensive algorithms and these are often subject of strict timing constraints imposed by the physical environment. In other words, embedded systems, as a compound of computational resources and software running on them, have to react to input signals within tight timing bounds in order to fulfill some stringent tasks such as the actuation of vehicle brakes or the deployment of airbags. For such systems designers must guarantee in advance that timing constraints will be fulfilled at any time during operation [1]. Therefore, performance analysis methods play an important role in the design process of embedded real-time systems. Consequently, provinding appropriate performance analysis solutions for hard real-time applications mapped on multi-core architectures with shared resources is the main goal of this thesis.

In what follows, Section 1.1 takes a closer look at the components of the embedded multi-core architectures and identifies the handling of the timing behavior of the multi-core components as a key design challenge. Section 1.2 discusses how timing aspects are classically handled across the development process of embedded real-time systems and identifies the lack of corresponding solutions for the new multi-core architectures. Finally, the contributions and the outline of this thesis are formulated in Section 1.3.

[1] In case the violation of time constraints [140] implies a major system malfunction with severe physical and economical consequences, the systems are called hard real-time systems, and soft real-time otherwise.

1.1 Embedded Systems: Multi-Core Architectures and Corresponding Design Challenges

In the previous section we highlighted the growing complexity of embedded systems in different application domains and provided some examples from the automotive industry. Without loss of generality, from now on we will continue focusing on the automotive domain, the automotive specific problems and solutions widely corresponding to other application domains.

The main reason for incorporating multiple cores on a single chip is to raise performance through parallel processing while saving costs and meeting the same thermal characteristics as the single-core processors. The following three use cases, illustrated also in Figure 1.3, are mainly driving the adoption of multi-core architectures for automotive electronic control units (ECUs):

1. the need to reduce the number of ECUs triggers the aggregation of multiple smaller ECUs into one multi-core ECU. This means that previously distributed software applications will be clustered into a single chip.
2. the need to integrate more features on the ECUs automatically require more processing power. For example in relatively high-performance domains such as engine control or advanced driver assistance systems more and more functions have to be implemented. In this case, multi-core architectures can be used to parallelize complex computations over multiple cores while enabling the integration of additional software functions.
3. the need to ensure high performance combined with high reliability, i.e. redundancy in case of system failure. This can be achieved by running the same software on distinct cores, e.g by running the cores in lockstep mode [66].

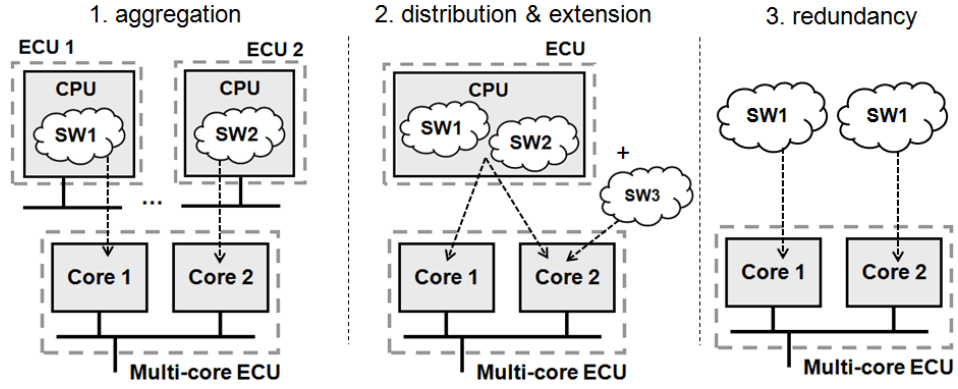


Figure 1.3: Multi-core systems - use cases

To enable the above mentioned use cases, the next generation of computational units of the automotive architectures will incorporate at least two processing cores (also called central processing units - abbreviation CPU). As an example, Figure 1.4 presents the block diagram of a multi-core architecture currently offered by Infineon [66, 67].

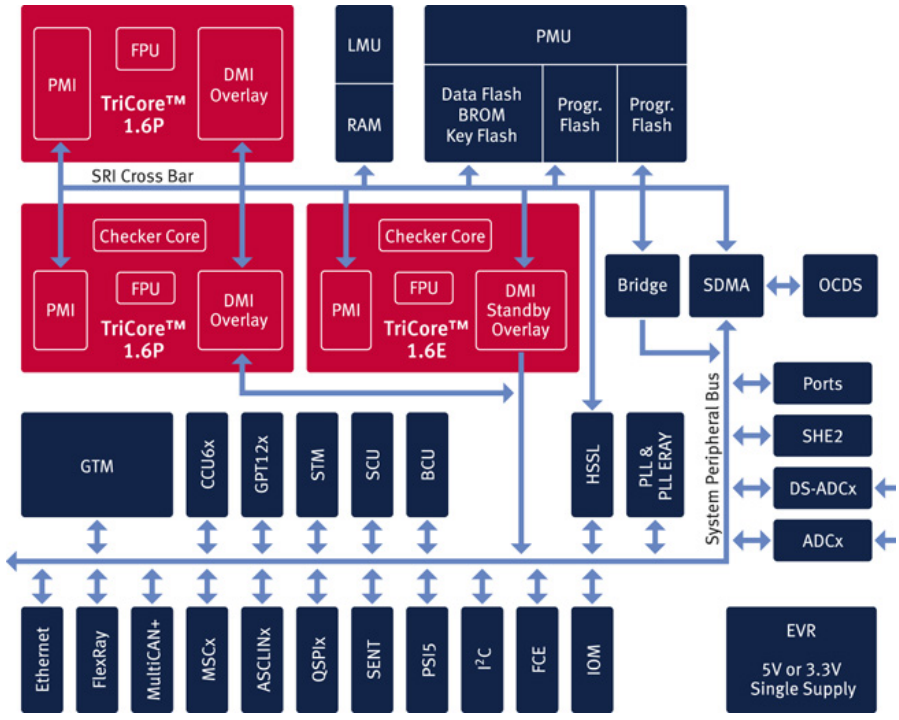


Figure 1.4: Block diagram Infineon Aurix multi-core architecture (Source [66])

The AURIX platform, the new multi-core solution developed by Infineon, was designed to fulfill the automotive needs with respect to performance and safety. The multi-core architecture is based on three independent 32-bit processing cores, with two homogeneous cores TriCore 1.6P running up to 300MHz and one core TriCore 1.6E running up to 200MHz, all three cores operating in the full automotive temperature range [66]. To meet the stringent safety requirements asked by the recent automotive safety standard ISO26262 [138], two out of the three cores can be configured in the so called lockstep mode. When running in the lockstep mode, a core is executing the same computational operation on the same data as the master core. This enables the comparison of the results computed by the master and the lockstep core and helps identify an incorrect behavior at runtime caused e.g. by hardware failures.

The three main cores are equipped with a local memory and can also access diverse shared resources, such as the Flash and the SRAM memory units or the bridge to the peripheral bus and therewith to the external I/O devices. In addition to the main computational cores, other controllers such as the Ethernet or FlexRay controllers may share the same memory units.

While the layout of the next generation automotive multi-core hardware architectures is to a large extent fixed with respect to processing units and shared resources, the major design challenges are more and more related to the software that has to exploit the multi-core components. The software infrastructure is essentially based on the existing hardware infrastructure and the software applications, that implement the desired

functionalities, will execute on the available hardware and software infrastructure. In this context system designers face the problem of deploying software applications, most of them developed, extended and often highly optimized over the years for single-core processors, to the different cores of a multi-core system.

The resource sharing challenge. One of the main challenges in making software applications compatible with multi-core systems is the high degree of complexity reached by the common use of shared resources, i.e. of the above mentioned shared memories, I/O devices and coprocessors or of logical data structures protected by semaphores.

Traditionally, safe sharing and communication between individual automotive tasks, i.e. elements of software applications, are realized by using global variables located in shared memories. These variables are accessed by using locks administered according to suspension-based or spinning-based synchronization protocols that are supported by the operating systems, as for example the suspension-based Priority Ceiling Protocol (PCP) supported by the OSEK/VDX OS [100] and the AUTOSAR OS [12]. The problem of concurrent accesses in single-core real-time systems is safely handled by these operating systems by using a combination of shared resource synchronisation mechanisms with a priority based processor scheduling strategy. Based on the relative priority of two tasks mapped on a single-core processor one knows which task interrupts the other one and how they interact with each other.

This is not the case anymore in multi-core setups where the common use of inter-core shared resources introduces an additional level of arbitration beyond that of the local processor core. There, a highly critical high priority task which runs on one processor core can interact in an unwanted way with a lower critical low priority task which runs on another processor core.

As an example consider the setup in Figure 1.5a) that illustrates the mapping of three tasks τ_1, τ_2 and τ_3 on two cores of the AURIX processor architecture in Figure 1.4. These tasks are assumed statically assigned to the cores and scheduled according to the static priority preemptive scheduling policy. Task τ_1 is assumed to have the highest priority and τ_3 the lowest. During execution these three tasks make use of a common shared resource, denoted SR, which can be exclusively accessed. Furthermore, the processor cores are assumed stalled for the time a task is waiting to get access to that shared resource or is holding it.

Figure 1.5b) depicts two possible runtime scheduling examples. The upper part of the figure provides a scheduling and shared resource access example where the shared resource SR is exclusively available to the tasks on Core 1. This execution corresponds to a single-core setup. In this example task τ_1 is assumed to make three accesses to SR and after its completion the lower priority task τ_2 starts executing and accesses the SR two times before completion. The bottom part of Figure 1.5b) illustrates a scheduling example for the case the shared resource SR is also accessed by the lower priority task τ_3 on Core 2, a situation that corresponds to a multi-core setup. The figure shows a worst-case scheduling situation where the execution of task τ_1 and τ_2 is delayed whenever the requested shared resource SR has previously been locked by the lower priority task

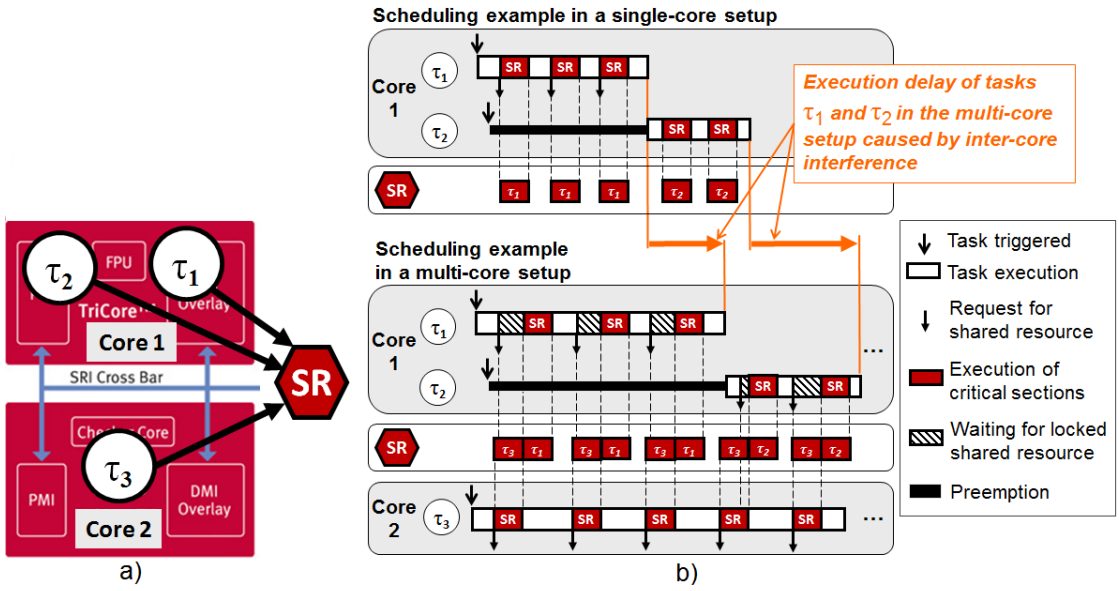


Figure 1.5: a) Tasks statically mapped on different cores share a common resource SR; b) Single-core vs. multi-core execution: Conflicting accesses for inter-core shared resources delay the completion of higher priority tasks.

τ_3 on Core 2. Each time a task is waiting to get access to SR the host core is stalled, which increases the task execution times. In this way, a low priority task on one core can slow down high priority tasks on another core and therewith delay their completion time. Such a delay can eventually lead to the violation of timing constraints. For hard real-time systems such a behavior can have severe consequences at runtime and therefore must not be left undiscovered at design time.

From a design perspective, the inter-core interaction via shared resources generates a timing interdependency between different tasks running on different cores, interdependency that has not only a negative influence on the computational performance [125, 124] but also challenges the predictability of the timing behavior [157, 130, 93].

The execution of the different tasks on the individual cores and therewith their requests for the shared resources are highly dynamic and independent of each other, fact that makes the prediction of the runtime inter-core interference difficult to achive at design time. Even more, because the software in the automotive industry is typically developed in a distributed development process (which will be discussed in Section 1.2) the above exemplified inter-core interference can be investigated only when the software pieces provided by different suppliers will be integrated on the same platform, which means relatively late in the development process. Furthermore, depending on projects and on the available architecture variants, different software components will be integrated differently which leads to a high diversity of multi-core setups that cannot be manually handled. A "general valid" mechanism that can handle the inter-core interference independent on the integration variant is highly required.

To ease the integration on automotive E/E architectures, the AUTOSAR standard [8] specifies a standardized development methodology and a software architecture which includes a runtine environment (RTE), standardized component interfaces and configuration files for basic software (BSW) and for application software components (SW-C), which communicate over a virtual function bus (VFB). All these finally enable the mapping and the integration of software supplied by different vendors to ECUs. With respect to multi-core architectures, with version 4.0 the AUTOSAR OS specification [12] started to standardize the support for distributed execution of software on embedded multi-core processors. More exactly, in order to handle the above discussed inter-core interference the AUTOSAR OS specifies a spinning-based inter-core shared resource synchronization mechanism. However, the current support for handling the functional aspects in the multi-core context does not implicitly solve the timing issues exemplified above, which are considered a non-functional aspect. Therefore, in order to ensure a safe application of multi-core architectures in real-time systems, appropriate solutions to investigate the impact of sharing resources on the performance and on the timing behavior of multi-core applications are required at design time. Intensive work of the AUTOSAR timing group and other industry-driven research projects (e.g. TIMMO-2-Use [151]), with the scope to develop a formal language and a methodology for timing and performance design in the automotive domain, indicate the industry's awareness of these issues.

The multi-mode behavior challenge. Another significant challenge in designing multi-core real-time systems arises when the applications, that have to be accommodated on the multi-core platforms, exhibit a multi-mode behavior at runtime. Acting in an complex environment that consists of diverse physical elements (e.g. natural environment, infrastructure, transportation) and often of humans participants, many real-time embedded systems are changing their functionality or characteristics over time due to changes in the environment or inside them. Such systems are called multi-mode systems and the applications running on these are called multi-mode applications.

Concrete examples of multi-mode systems are adaptive control systems in the avionic or automotive domain. These systems implement multiple operational modes and switch between them at runtime in order to respond to changing conditions in the environment, to switch to an emergency state or to change their resource usage. Beside the implicit need for an adaptive behavior of such systems, another important reason for implementing different operational modes is to save costs by integrating an increasing amount of applications (i.e. tasks of software applications) on a reduced number of computational resources (i.e. processors/processor cores). Obviously, the processing unit of any embedded system cannot be loaded more than 100% (in theory; in practice the threshold is lower due to feasibility reasons) and therefore one has to ensure that the tasks that can ever run on that processing unit are never requesting the processor capacity at the same time. In order to limit the maximum load on a system, multiple operational modes have to be defined and configured to exclusively make use of the available resources.

In practice, each mode has associated a specific set of tasks, which implement the mode specific functionality, and mode change protocols are responsible for managing the transition between modes. During a transition between two modes, some tasks can be

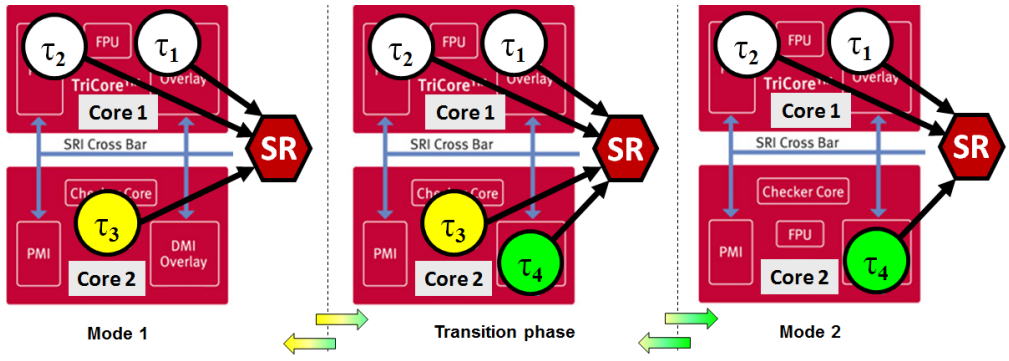


Figure 1.6: Task mapping in individual modes and during the transition between them.

stopped or simply aborted, new tasks can be activated or, in case there are multiple processing resources, some tasks can be migrated. Additionally, in many embedded systems there are tasks that cannot be stopped and must reliably execute in each operational mode and during the transition between modes.

As an example consider the setups in Figure 1.6, which illustrate the mapping of tasks τ_1, τ_2, τ_3 and τ_4 on two cores of the AURIX processor architecture in Figure 1.4 in two operational modes, denoted Mode 1 and Mode 2, and during the transition phase between them. The functionality in Mode 1 is implemented by the execution of the tasks τ_1, τ_2 and τ_3 whereas the functionality in Mode 2 is implemented by the execution of the tasks τ_1, τ_2 and τ_4. In this example task τ_1 and τ_2 have to continuously execute, independent on the mode change. Note also the fact that all these tasks can access the common shared resource SR. Furthermore assume that bidirectional transitions (illustrated by the arrows on the bottom of Figure 1.6) between these modes are implemented such that during one transition phase the execution of one of the tasks τ_3 or τ_4 has to be stopped, whereas the execution of the other one has to be started, and vice versa. Depending on the employed mode change protocol the execution of the tasks τ_3 and τ_4 on Core 2 during the transition phases can be exclusive (under so called asynchronous mode change protocols) or can overlap (under so called synchronous mode change protocols) [118]. On one hand, the exclusive execution of mode dependent tasks implies a low responsiveness, which is not acceptable for urgent activities. On the other hand, the simultaneous execution of tasks which belong to distinct operational modes during the transition phases can lead, even if only for a short time interval, to a temporary increased workload on the processors and to an increased inter-core interference [2]. Both effects can delay the completion of the tasks on thus lead to deadline misses. Therefore, when multi-mode applications are part of hard real-time embedded systems, designers have to ensure at design time that timing constraints are met at runtime under all circumstances. This means that timing requirements have to be fulfilled not only in the steady operational modes, when

[2] In comparison to the individual operating modes, accesses to the shared resource SR of the higher priority tasks on Core 1 are delayed during the transition phase by both lower priority tasks on Core 2 and not only by one of them.

all system characteristics are stable, but also when they are changing and the systems execute transitions between modes [118, 65, 145, 89].

Furthermore, a critical question that must be answered when designing multi-mode systems is: when has a system reached a steady state corresponding to one operational mode after a mode change? The overlap of multiple mode changes would make the execution of real-time systems completely unpredictable and has to be avoided at run-time. In order to guarantee that a mode change has completed and a successive one can be safely initiated, the duration of the transition phase, called settling time of a mode change or mode change transition latency, has to be known. Hence, obtaining this information is key in order to ensure the predictability of multi-mode real-time systems.

Similar to the support for multi-core technologies, recent specifications of the AUTOSAR standard provide system designers functional support for mode-management in automotive systems [10]. However, as discussed above in this chapter, the support for handling functional aspects does not implicitly ensure the correct and safe functionality with respect to non-functional aspects, i.e. a safe and predictable timing behavior, especially if shared resources are implied. Consequently, it is essential to provide designers of multi-mode real-time systems appropriate methods for timing and performance verification. However, in comparison to static multi-core applications, as considered in the example illustrated in Figure 1.5, if multi-mode applications share elements of a multi-core platform their performance and their timing behavior becomes even more difficult to capture [91, 92]. The dynamism of the tasks' execution and of their requests for shared resources is given not only by the processor scheduling policy and the shared resource arbitration strategy but also by the mode-management. Consequently, timing and performance verification instruments have in this case to jointly handle (i) the multi-core scheduling, (ii) the shared resource arbitration and (iii) the mode management, in order to enable safe predictions at design time.

Putting all together, the integration of static and multi-mode software applications on multi-core architectures consists of the challenging task to efficiently accommodate existing single-core processor software applications and new generation applications, being at the same time aware of the significant impact of the inter-core interference, caused by the competition for shared resources, on the applications' timing behavior. Hence, the safe and efficient design of multi-core real-time systems requires practical solutions for timing and performance verification.

1.2 Handling Timing Aspects in the Development Process of Embedded Real-Time Systems

1.2.1 Timing-Aware Development Process

The development process in the field of systems engineering is mainly based on the so-called V-Model [149]. The model, illustrated in Figure 1.7, consists of two branches which together provide a complete methodology to specify, design, detail, implement, integrate and validate a system.

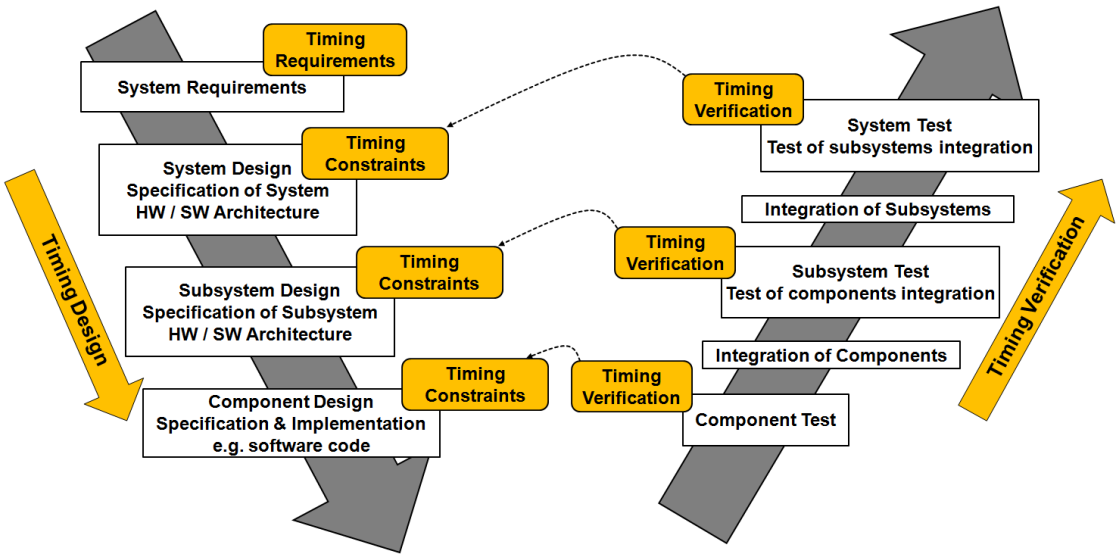


Figure 1.7: Timing aspects in the system development process according to the V-Model.

The model is especially used in the automotive industry where systems are traditionally developed in a complex distributed process which nowadays involves car manufactures (also called original equipment manufacturers, abbr. OEMs) and multiple tier-1 and tier-2 suppliers [28]. In this development process, OEMs are mainly responsible for the specification, design and integration steps, only rarely developing system parts in house. The development and implementation of the subsystems is typically outsourced to tier-1 suppliers which may further outsource parts to tier-2 suppliers. More exactly, during the early design phases of this development process decisions regarding the hardware and software architecture are taken. Further, the specifications of the overall design are split and detailed into specifications of subsystems and later into specifications of the subsystems' components. Whereas the separation of concerns leads to an increased efficiency of the development and verification of subsystems and components, it also implies a difficult process of integration and verification at system-level. Difficulties arise mainly because suppliers develop components and subsystems, based on the requirements specified by the OEMs, however often independent from each other and without considering the interoperability of the different parts. The task of the OEMs in this context is to ensure the interoperability of the multiple components when integrated into a subsystem and of the subsystems when integrated into the complete system and therewith the functional correctness of the entire system.

However, beside the functional correctness, there are other properties that have to be considered in order to guarantee the correct and safe functionality of complete systems. Such a property is the timing. As discussed earlier in this section, many (car) functions have to fulfill timing constraints in order to work properly, e.g the brakes of a car which have to be actuated immediately after pressing the brake pedal or the airbags which have to deploy instantaneously in case of a crash.

Two questions arise when addressing timing aspects in the development process, namely in which phases and how? As shown in Figure 1.7, similar to the functional aspects, timing concerns have to be first handled at design time and formulated in form of timing requirements/constraints and later verified in order to provide guarantees that the implemented solutions satisfy the initial requirements.

Timing design. More exactly, along with the functional aspects, timing concerns will be first specified as requirements at the system level by a system designer. The source of the timing constraints are usually the functions of the systems which are correlated with the systems physics and customer requirements and which finally enable the translation of terms such as "instantaneous" or "immediate" in time units, e.g. the brakes of a car shall be actuated not later than "x" ms after pressing the brake pedal. Further, as the overall system is split in subsystems, the system timing requirements will be also broken down and assigned in form of timing constraints to subsystems (e.g. ECUs, cores of a multi-core ECU or communication buses). Timing constraints on the different subsystems will be further decomposed and assigned to components (e.g. software applications). The specification of the timing behavior at different stages can be done for example with the so called timing description languages (e.g. TADL - Timing augmented description language [151]) or with the Timing Extensions of the AUTOSAR standard [9] supported with the release 4.0 published at the end of 2009.

Timing verification. In a distributed development process each supplier is responsible for the development of a specific subsystem (e.g. a single ECU) or of a software application and therewith it is also responsible for its timing behavior. In this context, timing verification plays an important role already at the components (software components) and subsystems' level (ECUs, buses). However, the timing behavior of individual components and subsystems is not independent, but is influenced and is influencing the timing behavior of other components and subsystems. For example, in case of multi-core ECUs, different suppliers will contribute parts of the software in form of AUTOSAR software components equipped with well-defined interfaces. As highlighted in Section 1.1, when integrated into a multi-core ECU, the timing of individual components is not independent but interacts at the core level through concurrent execution requirements and at the ECU level through the common use of shared resources. Thus, each individual component and subsystem is part of the overall system timing behavior which however can be completely verified by the system designer only in the late stages of the development process (see Figure 1.7), i.e. after integration, when all dependencies can be taken into consideration.

Therefore, the integration steps in a timing-aware development process have been recognized as significantly challenging [121, 122, 131] and demand for appropriate system-level timing verification instruments.

Highly relevant for the practical applicability of timing verification in the development process is its benefit. A late timing verification in the overall development process often means that even if problems are identified many design decisions can not be changed anymore or that the involved costs are huge. To cope with similar issues on the functional side, the design and development process in the automotive industry is vastly based on

design reuse. Software as well as hardware components from the most recent product are usually taken as a starting point for the next generation product. In this context, late stage timing verification can be transferred to earlier stages of a new generation product in order to help optimizing, modifying or extending a known system or to explore completely new design options. Automotive specific use cases from different application domains (ECUs, bus and network) have shown that knowledge gained from late verification can be applied to early design of new systems [122].

Still, independent on the application stage, a timing verification of complete systems is mandatory and requires appropriate solutions to uncover possible hazards that results from the complex timing interdependencies of the different systems' parts when integrated on the same infrastructure.

1.2.2 Current Practice: Solutions to Handle and Verify the Timing Behavior

Orthogonalization. A common solution for handling the integration challenges caused by the complex timing dependencies at system-level is the orthogonalization of system resources. For example, a system crossbar can be used for a spatially orthogonalization or buses and shared resources [102, 17, 7] may be assigned to the different processors in alternation according to a time-driven schedule. As this schedule will be independent of the actual run-time behavior, each component can then be verified in isolation as a minimum service will be guaranteed at run-time. Time-triggered architectures [73] are employed for example in the automotive domain for the static segment of the FlexRay [47] communication protocol or in the avionic domain with the TT-Ethernet [72]. A similar procedure is adopted in the avionic domain for a strict partitioning of processor resources [1]. While the orthogonalization of system resources simplifies the verification procedure, it also implies a conservative design with in general increased resource and possibly also power requirements. Furthermore, when applications exhibit dynamic properties such as varying bandwidth requirements, their behavior is very difficult to map to static schedules. To overcome an unnecessarily pessimistic design and to handle applications with a complex dynamic behavior a mixture of time-triggered and dynamic scheduling is preferable in practice as indicated by the FlexRay communication protocol that specifies a static and a dynamic communication segment [47].

With respect to multi-core processors, virtualization ensured by a hypervisor task will be used for enabling a functional separation between the different partitions, e.g. consisting of one or multiple cores [66]. However, while virtualization works for functionality, physically sharing chip resources still introduces non-functional, i.e. timing, dependencies between previously isolated task executions. Thus, timing aspects remain a major problem and have to be carefully investigated.

Simulation and measurement. In practice, simulation was and still is the predominant solution for investigating the systems' behavior. Simulation aims to investigate the systems' behavior based on hardware and software models, on different levels of abstraction, and on a set of input stimuli. This procedure allows debugging the functionality together with timing aspects for the common case. However, reliable verification of overall real-time properties is impossible, as the system would need to be subjected to an

exhaustive set of test patterns, which is difficult for larger systems [106].

Measurement is another timing verification solution used in the timing-aware development process. Timing measurement is essentially a tool supported analysis procedure used to collect timing information from real-time embedded systems at runtime, e.g. from the software running on a processor [128]. The collected timing information is compared with the specified timing constraints to verify whether these constraints are met by the implemented software. Gathering the timing behavior by measurement can be performed at the earliest when software code has already been written and flashed on the target hardware. Nevertheless the documented information is valuable and provides a clear image regarding the timing behavior of productive software, including available headroom for future software extensions. Combined with tracing, the measurement enables visualization of timing effects and help understanding and debugging timing issues.

Formal performance analysis. Another alternative for the investigation of the timing behavior is offered by formal analysis approaches. The general idea of formal methods is to determine conservative upper bounds on the systems' and system components' behavior, such as execution timing of software components on specific processors (i.e. worst-case execution times [158, 2]), response-times of individual software components on specific processors by accounting for local scheduling policies (i.e. worst-case response times [154]) or end-to-end latencies in case of distributed systems [64, 147].

To do that, formal methods are using (i) an abstract model of the systems that capture computational and communication demands of the software components, (ii) a set of mathematical equations that consider the resource sharing policies for processor scheduling (e.g. preemptive, non-preemptive), bus arbitration (e.g. non-preemptive or in a time-triggered fashion) and synchronization mechanisms for secondary resources (e.g. PCP [116]) and (iii) procedures that enable taking into account the interdependency between scheduling and communication at system-level in a holistic [152, 101, 110] or a compositional or modular way [120, 32].

Formal analysis methods have been proposed by academia starting with 1973 [79] and nowadays are gaining more and more attention also in industry [2, 147]. For example the SymTA/S analysis framework [147] has been used by Volkswagen Steering Systems for the ECU hardware selection of a new electromechanical steering system [122]. Daimler Research used the same analysis framework not only for analyzing and dimensioning individual buses but in the context of network topology design for next-generation car platform of Mercedes-Benz [83]. These commercial case studies, clearly show that tool supported methods that have been suggested in research for some time (e.g. code-level analysis [2], system-level schedulability and response-time analysis [147]) now become feasible to be used in actual productive environments.

With respect to static and multi-mode multi-core systems, various formal analysis solutions have been proposed over the years for multiprocessor and multi-core setups with and without to consider inter-core shared resources [4, 75, 132, 22, 156] and mode changes [118, 95]. The list of the related work in these fields is far much larger and will be covered in the next chapters of this thesis. Relevant for the moment is the fact

that the applicability of previous research in the development of multi-core real-time systems in the industry is still limited as many system details are not covered on the modeling and analysis side. Concrete examples are the lack of analysis approaches that can be used for upper-bounding the blocking times and the response times of multi-core applications under complex processor scheduling policies and inter-core synchronisation protocols, as supported today by the AUTOSAR OS, or the lack of a solution for the analysis of mode change transition latencies for multi-mode systems.

Hence, the current methodologies have to be extended to handle the earlier discussed design challenges namely, the complex timing behavior caused by the inter-core interference in case of sharing common resources and the dynamic behavior of multi-mode applications. Obviously, in order to facilitate an analysis, new models are required to capture as accurate as possible the various timing parameters of such systems. Further, analysis elements which exploit these models and system-level procedures which combine the analysis elements have to be provided. Also, highly relevant for the applicability of formal performance analysis to industry specific setups is their compliance with the specifications of the industry's standards, as for example the AUTOSAR specifications in the automotive domain.

1.3 Thesis Contribution and Outline

The previous sections highlighted the need of different embedded application domains for powerful multi-core architectures and the challenges associated with the design process of multi-core solutions. Because multi-core processors are often required to accommodate safety-critical applications with hard real-time constraints, designers have to deal with the difficult task to safely and efficiently accommodate these applications such that their timing constraints are never violated. One main challenge is given by the use of physically shared hardware (e.g. shared memories, I/O devices, coprocessors) and the synchronization via logical resources (semaphores) which introduce dependencies between task executions on different cores, thus jeopardizing the real-time behavior of the entire system [90, 131, 130, 93]. Another significant challenge arises when the real-time applications, that have to be accommodated on multi-core platforms, exhibit a multi-mode behavior at runtime. Their adaptive nature implies a complex timing behavior characterized by dynamic changes of the applications' timing properties at runtime. This dynamism makes the timing behavior difficult to capture [118, 65, 145, 89, 92].

To manage these problems, the design process of multi-core real-time systems requires adequate methods and tools for timing and performance verification. For this purpose, the present thesis (i) deals with the investigation of the timing behavior of hard real-time static and multi-mode applications which share resources in multi-core architectures and (ii) proposes corresponding tool supported formal performance analysis solutions. The context and the contribution of this thesis are summarized in what follows.

- Chapter 2 provides the fundamentals of system-level performance analysis methods for distributed and multi-core real-time systems including their underlying system models. Furthermore, for the scope of this thesis the general compositional

system-level performance analysis procedure and its dedicated extension for multi-core systems with shared resources are detailed. The modeling and the analysis approach represent the foundations for the new solutions contributed by this thesis for the analysis of real-time single-mode and multi-mode applications mapped on multi-core systems with shared resources.

- Chapter 3 focuses on the timing analysis for multi-core systems with shared resources. One of the major concerns of the industry's activities towards multi-core solutions is related to the implementation of scheduling policies and shared resource synchronization mechanisms that best match the practical requirements.

  For this purpose, Chapter 3 highlights key components of a safe synchronization protocol for shared resources in multi-core systems and investigates the impact of different design decisions with respect to shared resource arbitration and multi-core scheduling policies on the multi-core systems' timing behavior.

  In order to evaluate the different design options, new analysis approaches are proposed for deriving blocking-times and response-times for multi-core applications under different scheduling policies and shared resource arbitration strategies. The new methods can be integrated into the general compositional system-level analysis procedure discussed in Chapter 2 and, unlike existing analysis approaches, these cover realistic system configurations with tasks that exhibit arbitrary activations and deadlines and use a sophisticated model to capture the resource load and timing between individual requests for shared units. Furthermore, in comparison to existing analysis solutions, the proposed approaches are not limited to preemptively scheduled multi-core setups, they also handle non-preemptive scheduling in the context of multi-core systems. Also, the more complex setup consisting of the combination of preemptive and non-preemptive scheduling is covered. This extends the applicability of formal performance analysis to industry specific setups, as for example for AUTOSAR conform automotive multi-core controllers where preemptive and non-preemptive scheduling will co-exist on each core [12].

  Experimental evaluations highlight the difference between the different scheduling and shared resource synchronization options, confirm the benefit of distributing the computational load across multiple cores and demonstrate the applicability of the proposed analysis solutions.

- Chapter 4 focuses on the timing analysis of multi-mode real-time applications. For that purpose the system model and the compositional analysis procedure discussed in Chapter 2 are extended to consider elements of multi-mode systems.

  Next, the settling time of a mode change, called mode change transition latency, is identified as an important system parameter that has been neglected before. Known approaches that address the problem of timing analysis for multi-mode real-time systems are restricted to applications without communicating tasks. Also, these assume that transitions between operational modes are initiated only during a steady state, however, without indicating when a system executes in a steady state. In this context, Chapter 4 contributes an analysis algorithm which gives a

maximum bound on each mode change transition latency of multi-mode distributed applications thereby overcoming limitations of previous work.

Further, the problem of accommodating multi-mode applications which share resources on multi-core systems is considered. Various mode change and resource arbitration protocols, and corresponding timing analysis solutions were proposed for either multi-mode or multi-core real-time applications. However, no attention was given to multi-mode applications that share resources when executing on multi-core systems. This subject is addressed in Chapter 4 in the context of automotive multi-core processors which use the AUTOSAR specifications for partitioned multi-core OS [12] and the guidelines for mode management [10, 13]. An approach for safely handling shared resources across mode changes in this setup is discussed and a corresponding timing analysis method is provided.

The applicability and usefullness of the proposed analysis solutions is demonstrated by experimental data and emphasized by an automotive specific use case.

Even if this thesis mainly focuses on the automotive domain, one of the main technology innovation drivers worldwide, the addressed problems and the proposed solutions widely correspond to other application domains.

2 System Modeling and System-Level Performance Analysis

This chapter provides the fundamentals of system-level performance analysis methods including their underlying system models. First, approaches existing in literature are surveyed. Then, a general modeling approach for systems is introduced, approach that is further particularized for multi-core systems with shared resources. Based on this, the compositional performance analysis in the presence of shared resources is detailed. The modeling and the analysis approaches represent the foundations for the new solutions contributed by this thesis for the analysis of real-time single-mode and multi-mode applications when mapped on multi-core systems with shared resources.

2.1 Survey on System-Level Performance Analysis Approaches

Three main classes of system-level performance analysis approaches can be distinguished in literature: (i) simulation-based, (ii) timed-automata and (iii) classic real-time system theory with the holistic and compositional system-level extensions.

Simulation. As briefly discussed in the introduction in Section 1.2.2 extensive simulation is widely used in practice for investigating the systems' behavior. Based on hardware and software models on different levels of abstraction and on a set of input stimuli simulation aims to investigate the systems' behavior. However, simulation-based analysis solutions are accompanied by a challenging trade-off. The more precise the systems are modelled, the accurate will be the derived systems performance characteristics and the more test cases are covered the more confidence will system designers have in the correct runtime behavior. However, detailed systems' modeling and exhaustive test case coverage automatically imply higher complexity, higher computational demands and therewith long analysis times. All these make the simulation-based verification process expensive. To cope with this issue, simulation is often limited to a relevant but particular set of test parameters, which however leads to an insufficient corner case coverage. This means that at runtime there might be situations represented by a specific combination of the system's internal status and external influence that was not covered during simulation and in which the systems requirements are not fulfilled anymore. Even if practical experience shows that simulation has been successfully applied for many years, the severe specifications of the safety regulations and standards require more and more the exhaustive operational states coverage.

Timed-Automata and Model Checking. Another approach for the specification and analysis of real-time systems is based on timed-automata and model checking [63, 14, 39]. In this case, the analyzed systems are first modeled using timed

automata. Then, based on the systems' formal models, a model checker (e.g. UPPALL [39]) performs a reachability analysis to verify whether the system adheres to the specific timing properties. Timed automata and model checking based approaches have been proposed for distributed systems [63] and newly also for multiprocessor systems without shared resources [59, 27] and with shared resources [82, 55]. Whereas model checking is able to take many global dependencies into account and thus to provide tight performance bounds, it also implies an exhaustive state space coverage and therewith a significant analysis effort which does not scale well with system size and heterogeneity [59]. In other words timed-automata and model checking based formal analysis may require long or even unbounded verification times [106]. Recent research on model checking techniques for timing analysis highlights the same scalability problem [82, 55] and invests effort in alleviating the state-explosion problem [55] or in adapting the reachability algorithms to take advantage of the available parallelism on modern multi-core processor architectures [38].

Holistic approaches. Similar to timed-automata and model checking based solution, holistic formal analysis approaches handle the timing analysis problem by considering a global view of the systems. The analysis principle is however different. Holistic performance analysis approaches essentially extend the classical uniprocessor scheduling theory (e.g. worst-case response time analysis for static priority preemptive scheduling, abbr. SPP) toward distributed systems by considering specific combinations of input event models, resource sharing, computation and communication policies (e.g. based on time division multiple access, abbr. TDMA) [152, 101, 111, 56, 110]. Whereas holistic approaches efficiently exploit global system dependencies in order to enable tightly calculated timing bounds and reduced analysis effort, they are difficult to be used for large (i.e. with many components) and heterogeneous (i.e. with different scheduling policies) systems. Furthermore, due to their "holistic" nature which requires taking all systems dependencies into account for each specific system setup, they are hard to be used in the context of today's distributed system development process (see Section 1.2).

Compositional and Modular Performance Analysis Approaches. Given the heterogeneity of modern embedded system architectures, a different type of analysis approach, called compositional or modular performance analysis [120, 119, 32, 121, 64, 69], was developed to cope with the performance analysis of arbitrary complex architectures. The main idea of the compositional and modular performance analysis solutions is to:

(i) break down the analysis complexity of large systems consisting of multiple processors interconnected by buses into separate system components analyses (i.e. individual analysis of each processor and bus) which can be easier handled and

(ii) to compose the results of the individual component analyses by a system-level analysis procedure in order to derive the system-wide timing behavior.

In case of the compositional system-level analysis procedure [121, 64], the local component analyses are essentially schedulability analysis algorithms dedicated to different types of resources (uniprocessors, CAN-buses [42], FlexRay-buses [97]) scheduled according to different resource arbitration policies known from the scheduling theory, e.g.

for static priority preemptive (SPP) [154], static priority non-preemptive (SPNP) [42] or round-robin (RR) [114]. In the context of the system-level analysis procedure the performance characteristics (e.g. worst-case response times) computed locally for each individual component are propagated through the system by the use of the so-called *event models* [58, 120, 119, 121] and thus applied in a compositional manner.

Parallel to the Compositional Performance Analysis (CPA) approach, the Modular Performance Analysis (MPA) has been developed [32, 69]. The MPA framework is based on Real-Time Calculus (RTC) [150] and relies on a compositional methodology which has its roots in the analysis composition employed by Network Calculus in order to derive worst-case bounds on communication networks [36, 37, 77]. Similar to the event models used by CPA, MPA uses a concept called *arrival curves*, which can be seen as a generalization of the standard event models in [121]. In addition, MPA uses the notion of *service curves* to represent the computational and communication capabilities of the systems' resources (i.e. processors and buses). In the system-level analysis procedure these curves are used at the resource level to derive remaining computational service after servicing the applications computational demands, information which is further propagated across the different system resources in order to derive the system-wide system behavior.

Through their compositional or modular analysis principle, the above discussed approaches are well suited for today's distributed development process. Therewith the analysis of different subsystems, often developed independent from each other and integrated in multiple different systems, can be simply embedded into the general system-level analysis procedure in order to verify the systems' overall timing characteristics. Beside flexibility and scalability, these approaches are also characterized by short analysis times which make them suitable even for the analysis of large systems.

Furthermore, the compositional performance analysis procedure was extended in the last years to account for the complex timing dependencies that arise in multi-core setups due to the use of inter-core shared resources e.g. in [90, 130, 132]. Thus, for the scope of this thesis, the compositional performance analysis methodology and its multi-core aware extension will be detailed (see Section 2.3) and used as a basic building-block for the analysis solutions proposed in this thesis.

In order to do that, the general underlying system model of the compositional performance analysis and of the analysis solutions proposed in this thesis will be next introduced. This system model will be refined across the next chapters to consider more detailed system properties which will be finally considered to capture more exactly the complex timing behavior of single-mode and multi-mode real-time applications when mapped on multi-core systems with shared resource.

2.2 System Model

In order to reason about the runtime behavior of a system, the system functionality is verified based on an abstract model. In what follows, a general modeling approach for real-time systems is introduced along with the corresponding terminology.

2.2.1 General System Model

In general, despite the way of doing the analysis, system-level analysis procedures, as the ones mentioned in Section 2.1, share a common underlying way of modeling systems and capturing their properties. Thus, a system model consists in general of the platform (hardware) model description, of the application (software) model description and of the mapping description, i.e. information regarding the assignment of the application model elements on the platform model elements. Further, in order to reason about the timing behavior of a system the system model is augmented with timing properties. These aspects will be detailed in what follows.

A **platform model** $\mathcal{PM}$ describes a finite set of platform elements PE consisting of a finite set PE_{comp} of computational resources (i.e. processors), a finite set PE_{comm} of communication resources (i.e. buses/networks), a finite set of PE_{shared} of secondary shared resources (e.g. shared memories) along with connectivity information and parameters such as processing power, scheduling policies, shared resource arbitration strategies or protocol specifications.

An **application model** $\mathcal{AM}$ describes a finite set of applications $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ ($m \in \mathbb{N}$), where each application A_i ($A_i \in \mathcal{A}$) typically consists of a finite set of computation and communication tasks $\mathcal{T}_i = \{\tau_1, \tau_2, \ldots \tau_n\}$ ($n \in \mathbb{N}$).

Applications are typically represented as directed task graphs, where the nodes of the graph represent the computational or communication effort of the application and the edges represent functional dependencies between the nodes. For the purpose of this thesis only applications expressed as directed acyclic graphs are considered.

To express the influence of the environment on the applications, the task graphs contain another two special elements namely, *sources* which are tasks that are the first in a task chain, and *sinks* which are tasks that are the last in a task chain [68]. Further, all edges in a task graph, i.e. including the edges from sources to the first task in a chain and from the last task in a chain to the sink, generally describe the internal and external connectivity and communication of the different tasks of an application.

The **mapping** of the applications to the platform is specified by a function $\mathcal{MAP}$ which assigns each computational and communication task of each application to a computational or communication platform element:

$$\mathcal{MAP} : \forall \tau_i \in \bigcup_{j=1..n} \mathcal{T}_j, \mathcal{T}_j \in \mathcal{A} \rightarrow PE_{comp} \cup PE_{comm}$$

Finally, based on the individual model elements above a system model is generally defined as:

Definition 2.1 (System Model) *A system model $\mathcal{SM}$ is a tuple $(\mathcal{PM}, \mathcal{AM}, \mathcal{MAP})$, consisting of a platform $\mathcal{PM}$, the applications $\mathcal{A}$ and the mapping information $\mathcal{MAP}$ of the applications' tasks to the platform.*

2.2.2 Timing Model

In order to reason about the timing behavior of a system, the elements of the system model above are enhanced with timing characteristics. Furthermore, the timing behavior of a real-time system is directly related to the scheduling policies and shared resource arbitration strategies. All these aspects will be covered in this subsection.

Timing Bounds. The applications' tasks are assumed as time consuming entities which require a certain execution time on the resources they are mapped to. The maximum execution time required by a task τ_i on a resource to complete its corresponding job (i.e. computation on a processor or transmission time on a bus) is called *worst-case execution time* (WCET) and is denoted C_i.

At runtime, tasks will be executed multiple times and each of these task instances is called a *job* and denoted with J. Thus, with J_i we denote a job of task τ_i. When speaking about the worst-case execution time C_i of a task τ_i we implicitly understand that C_i is associated with each job J_i of τ_i. Associated to real-time systems are also timing bounds in which the jobs of the different tasks have to complete their execution in order to ensure the correct system functionality. These timing bounds are called *deadlines* and denoted D_i.

Computational or communication resources assign service, i.e. execution time, to tasks according to a scheduling policy that is realized by a scheduler, which is part of the operating system. Very often scheduling policies for real-time systems are based on priorities, i.e. they assign processor time to tasks depending on their priorities. Therefore, each task τ_i has a *priority* associated, priority which we consider indicated by the tasks index i. These priorities can be assigned offline, i.e. statically at design time, in case of fixed-priority scheduling as employed by OSEK [100] and AUTOSAR [12], or dynamically at runtime in case of dynamic priority scheduling (e.g. EDF - earliest deadline first [29]). As dynamic priority assignment is not common in current safety-critical real-time systems, the analysis solutions provided by this thesis apply to systems with static priority scheduling.

With respect to multi-core setups, scheduling policies can be classified into two different classes, depending on the flexibility of scheduling any given task: the *partitioned* or the *global / non-partitioned*. In the partitioned scheduling approach tasks are statically mapped to the processor cores, which are separately scheduled at run-time. In case of global scheduling approaches, the scheduler maintains a single scheduling queue of tasks, from which tasks are dynamically dispatched on the available processor cores and possibly migrated during execution. Partitioned scheduling fits best the current practice in the industry and is therefore the main focus of this thesis.

During execution, in order to fulfil their jobs, computational tasks may request service not only from the resource on which they are mapped (i.e. resources in R_{comp}) but also from a secondary shared resource (i.e. a shared resource in R_{shared}). Typically, such a secondary shared resource is a shared memory, in which tasks write or read some global variables. Such a shared resource could be also a I/O device. For the purpose of this thesis shared resources are considered objects (e.g. data structures or

devices) that require serialized access for which purpose they are protected by locks (e.g. binary semaphores). Accesses to shared resources are arbitrated according to a synchronization policy such as Priority Ceiling Protocol (PCP) [100, 116] for single-core processors or its extension for multi-core processors the Multiprocessor Priority Ceiling Protocol (MPCP) [116].

There are two types of shared resources: local shared resources (LR) and global shared resources (GR). For simplicity we omit the word shared, when explicitly indicating a shared resource as local or global. Local resources reside on each core and can be accessed only by the tasks that are mapped to it. Global resources are assumed in a separate shared resource module and can be accessed by tasks mapped to different cores.

In case a shared resource has to be exclusively accessed, the execution of a task when accessing it is generally called *critical section* [1]. A critical section guarded by a semaphore and protecting a global or a local resource is called global critical section (*gcs*) or local critical section (*lcs*). The maximum number of global critical sections that each job J_i of a task τ_i executes before its completion is n_i^G, and ω_i^{GR} represents the maximum duration of such a global critical section. Correspondingly, ω_i^{LR} represents the maximum size of a local critical section when accessed by jobs of a task τ_i.

Note that for the scope of this thesis we assume that processors are considered to have a timing compositional architecture [157], which means that delays of tasks due to shared resource accesses are additive to the tasks' execution times.

Timing Events. The execution of a task in a real system is always the result of an activation *event*, which can be external or internal such as the arrival of an interrupt, the expiration of a timer or the result of task or bus communication being finished. For example, in the automotive domain many functions are executed cyclically on a processor or are cyclically transmitted over a bus. To capture this behavior, tasks in the model often have an *activation period* T_i associated. Furthermore, a common assumption in literature is that applications task graphs correspond to dataflow graphs described e.g. by a Kahn Process Network [71, 68]. Under this assumption edges of the task graphs correspond to communication channels with first-in-first-out (FIFO) buffer semantics. Thus, each task has one input FIFO buffer [2] associated from which it reads the activating data. In case of inter-task communication in which a task produces data for another tasks (i.e. task chaining), the execution completion of one task leads to the activation of another task, i.e. a task writes data into the input FIFO of a dependent task. Corresponding to the activation event and the associated task execution there is always a completion event which indicates the termination of the task execution. For the purpose of this thesis it is assumed that one input activation event produces one event on termination.

As can be seen, the notions of *events* and the *timing of events* capturing specific points in time corresponding to an action in the physical world are key in the timing analysis

[1] In practice a critical section is a piece of code that accesses a shared resource that must never be simultaneously accessed by more than one task.

[2] Elements in a FIFO buffer are processed strictly in order.

and are part of the specified system models [120, 32, 121, 68, 64, 132].

In literature, the activating events are assumed as captured by *event streams* and the behavior of the event streams is described using *event models* [58, 119, 32, 64, 136, 127]. One example is given by the *standard event models* [119, 121] which captures key properties of event streams using three parameters namely, the activation period P corresponding to the distance between events, the activation jitter J which indicates that periodic events can vary around their exact position within a jitter interval, and the minimum distance *dmin* between successive events within a burst.

This thesis follows the definitions in [121, 136, 132] with the observation that [136, 132] generalized the initial definitions in [121] to support *arbitrary event models* and not only *standard event models*.

In general, event models can be expressed with two types of functions namely, the event arrival functions and the event distance functions.

Definition 2.2 (Event arrival functions) *The upper and the lower event arrival functions, denoted $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$, specify the maximum and the minimum number of events that may occur in an event stream during any time interval of size Δt.*

$$\eta^+ : \mathbb{R}^+ \rightarrow \mathbb{N} \tag{2.1}$$
$$\eta^- : \mathbb{R}^+ \rightarrow \mathbb{N} \tag{2.2}$$

Correspondingly, event models can be expressed with the event distance functions:

Definition 2.3 (Event Distance Functions) *The minimum and the maximum event distance functions, denoted $\delta^-(n)$ and $\delta^+(n)$, specify the minimum and the maximum time intervals during which at least n $(n \geq 1)$ events may occur.*

$$\delta^- : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \tag{2.3}$$
$$\delta^+ : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \tag{2.4}$$

Figure 2.1 shows an example of the functions η and δ, example which illustrates that the functions η and δ are pseudo-inverse, i.e. can be converted to each other [132].

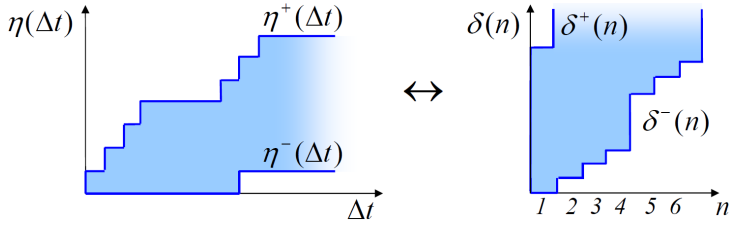


Figure 2.1: Event stream representation.

As discussed earlier in this section, during their execution tasks perform a number of requests for some secondary resources. Similar to the activation and termination of a

task these requests are associated with timing events. In order to capture the timing behavior of the shared resource requests [90] proposed a model which uses the event model concept above and the maximum number of requests issued by each instance of a task.

Definition 2.4 (Shared Resource Request Bound) *The* Shared Resource Request Bound $\tilde{\eta}_i^+(\Delta t)$ *is the maximum number of requests that may be issued by a task* τ_i *to a shared resource within a time window of size* Δt*.*

The computation of the shared resource request bound functions $\tilde{\eta}_i^+(\Delta t)$ is addressed in Chapter 3 in the context of the proposed timing analysis solutions. Until then, Figure 2.2 illustrates the execution of a task on a processor along with the corresponding timing events. The example shows the execution of two instances of a task τ_i triggered

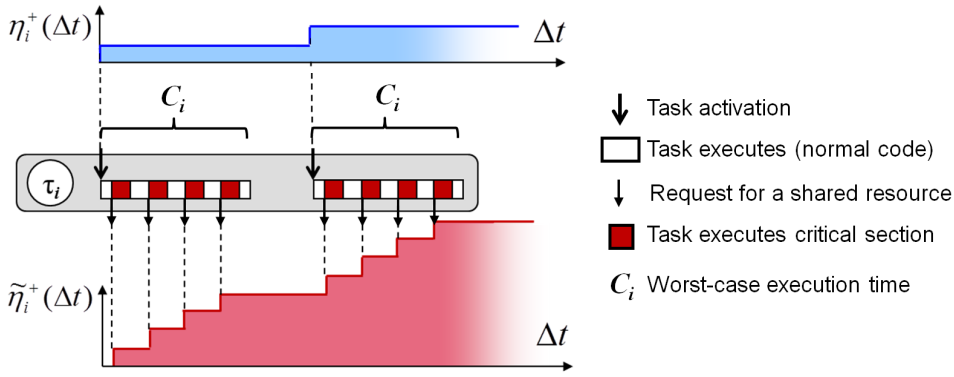


Figure 2.2: Example of task execution and the associated upper event arrival function η^+ and shared resource request bound function $\tilde{\eta}^+$.

according to the upper event arrival function η_i^+. Further, it is assumed that task τ_i performs during its worst-case execution time C_i four requests for a shared resource and the corresponding four critical sections are considered part of the worst-case execution time. In this case, the shared resource request bound function $\tilde{\eta}_i^+$ is given by the upper event arrival function multiplied with the number of requests per task instance.

2.2.3 System Model and Task State Model: Example

Figure 2.3 depicts a model of a dual-core system that may be part of a larger automotive system. The model covers both the general system model in Section 2.2.1 and the timing model in Section 2.2.2. Thus, the system consists of two processor cores, Core 1 and Core 2, on which there are three applications statically mapped. The first application consists of the tasks τ_1 and τ_2 connected by an edge indicating the functional dependency between them. The second application consists only of task τ_3 and application three of task τ_4. The sources of the three applications are indicated with the tasks So_i whereas the sinks are not illustrated. The edges between tasks and between sources and tasks are annotated with the functions η_i indicating the tasks' input event models (often called input activation models). Whereas the activations of tasks τ_1, τ_3 and τ_4 are produced by

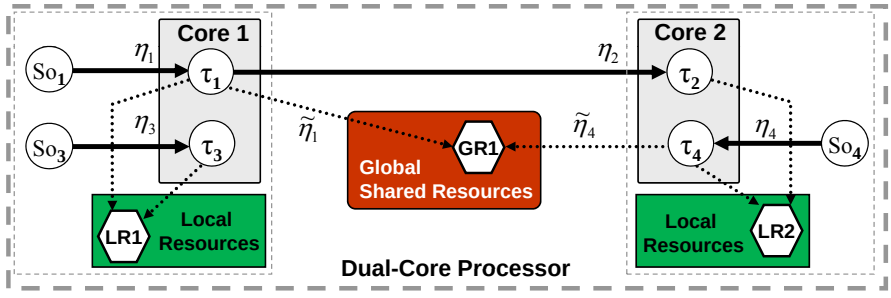


Figure 2.3: Example of a dual-core processor with tasks which access local (LR) and global shared resources (GR).

a source such as a sensor signal or the expiration of a timer, the input activation model η_2 of task τ_2 is given by the the output activation model of task τ_1. Task indices indicate their static priorities where lower indices mean higher priority, i.e. index 1 represents the highest priority in this system and 4 the lowest one.

The tasks in this setup access diverse secondary shared resources namely, tasks τ_1 and τ_3 share a local resource on Core 1 denoted $LR1$ and tasks τ_2 and τ_4 share a local resource of Core 2 denoted $LR2$. Beside the local resources tasks τ_1 and τ_4 share a global resource denoted $GR1$. The dotted edges between tasks are annotated with the functions $\tilde{\eta}$ which indicate the tasks' shared resource request bound.

As discussed earlier in Section 2.2.2, in order to execute, tasks mapped on processor cores require computational service and access to the secondary shared resources. Computational service is made available by a scheduler according to a scheduling policy which in case of real-time systems typically considers the priorities of the tasks. As an single processor core can always execute only one task at the same time and as several tasks will actually compete with each other for the processor core service the execution of multiple tasks on the same core is in general not independent. Furthermore, these tasks will also compete for the shared resources. The execution of tasks on priority based scheduled processor cores is captured by a task state model, where the model's states corresponds to the different states a task can experience at runtime.

For exemplification Figure 2.4 depicts the execution of an instance of task τ_4 on Core 2 and the corresponding task state model [3]. The task state model corresponds to the typical real-time task model widely used in literature [154] and industry [100]. The default state of a task is *suspended*, i.e. it doesn't execute. A task enters the *ready* state when it is activated and changes to the *running* state (i.e. the task will execute) when the scheduler starts it i.e. provides the required computational service. During execution, a higher priority task (e.g. τ_2 in our example) may be activated. In this case the so far running task is preempted and changes its state to *ready*. When a task tries to access a shared resource (i.e. performs a request for a shared resource) but this is currently

[3] In comparison to the OSEK basic task state model the OSEK extended task state model includes the waiting state to capture the blocking effects [100].

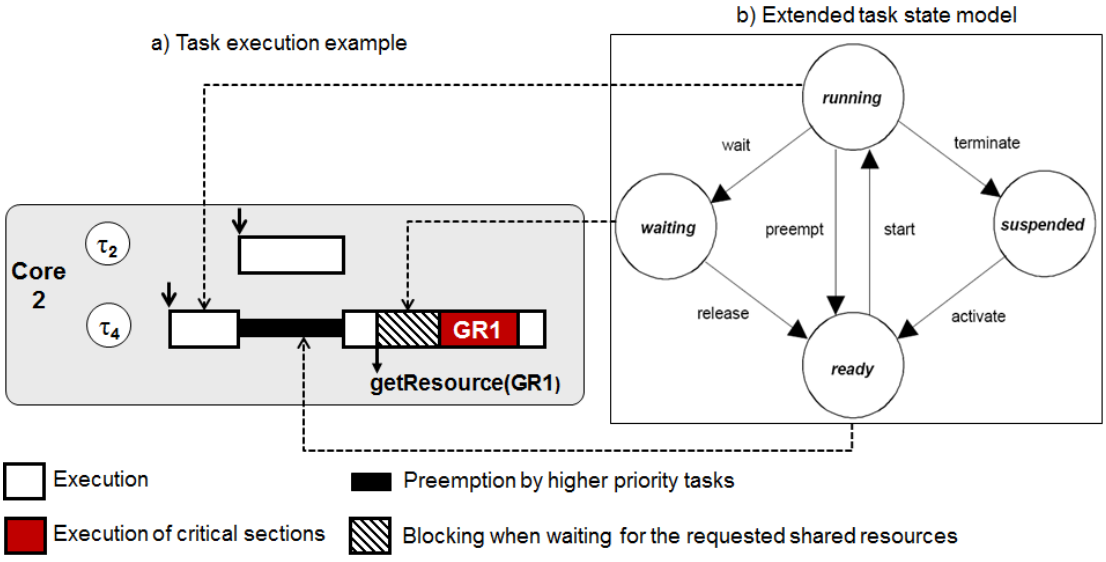


Figure 2.4: Example of a task execution and corresponding extended task state model (OSEK state model [100]).

not available the task changes to the state *waiting*. The task will switch to the *ready* state when the requested shared resource becomes available. The execution of critical sections and normal code is captured by the same state *running*. After termination the task switches to the default state *suspended*.

The graphical visualization of the system model, of the event arrival functions and task executions as in Figures 2.1, 2.3 and 2.4a) will be used across the next chapters when reasoning about the timing behavior of different tasks.

For such a complex system model, performance analysis in general and the compositional performance analysis in particular are concerned with the computation of the tasks *worst-case response times* (WCRT), i.e. for each task τ_i of the largest time interval between the activation of any of the τ_i's jobs J_i and the termination of the corresponding job execution. The main goal is that by handling the timing of events together with the associated task executions to provide accurate information regarding the completion times of the tasks execution. By comparing the completion times of the tasks' executions with the tasks' deadlines one can answer the question whether a system will always work correctly, i.e. whether a system will fulfil all its timing constraints. The answers obtained through a timing analysis based on the system models also hold for the runtime behavior of the physical system.

Note that, the system model introduced above considers the upper bounds on the tasks execution times i.e. the WCETs C_i as given. In practice, the derivation of tasks' worst-case execution times is a difficult problem which is subject of special formal analysis methods [158, 157] such as the static analysis of the tasks control flow which contains the logical structure of the task execution. Alternatively, extensive simulation is commonly

used in practice and provides in general sufficiently accurate estimations of the tasks execution times. However, simulation suffers from an insufficient corner case coverage i.e. may not always find the longest execution path of the investigated piece of code. The problem of deriving worst-case execution times is orthogonal to the problem of deriving worst-case response times. For the purpose of this thesis it is assumed that the considered hardware platforms are free of timing anomalies [158, 157], the worst-case execution times have been safely derived and these will be used for deriving timing bounds at system level.

2.3 Compositional System-Level Performance Analysis Procedure for Multi-Core Systems with Shared Resources

2.3.1 General Analysis Procedure

Relying on the system model introduced in the previous section this section presents the compositional system-level analysis procedure for multi-core systems with shared resource. The shared resource aware compositional analysis procedure for multi-core systems, developed over the last years and used in different multiprocessor and multi-core setups [135, 90, 130, 93, 132, 88], is based on the principles known from the compositional system-level performance analysis (abbr. CPA) for distributed and MPSoC (Multi-Processor System-on-Chip) systems [120, 32, 119, 64].

The basic idea of CPA is to break down the analysis complexity of complete systems into separate system components analyses and to interleave the analysis of individual components with the propagation of event models [120, 119]. The classic CPA procedure is illustrated in Figure 2.5a) and the extended version applicable for multi-core systems with shared resources in Figure 2.5b).The compositional system-level analysis is essentially an iterative procedure which works as follows:

I. First, the external activation patterns are derived from the environment (e.g. sensor sampling rates, maximum engine rpm, minimum human response time). The behaviors of the individual tasks are investigated in detail to gather all relevant data such as the best-case and worst-case execution times. As already mentioned in the previous sections, these can be derived with formal methods such as in [158], but extensive simulation is also common in practice.

II. The input event models captured by the functions η and δ are supplied to the individual components.

III. The input event models are then used to derive the behavior within individual components (such as a processor or a bus), accounting for local scheduling interference. This means that based on the underlying scheduling strategy as well as stream representations of the incoming workload modeled through its *activating* or *input event models*, local component analyses systematically derive worst-case scenarios to calculate worst-case (and best-case) task response times (BCRT, WCRT), i.e. the time between task activation and task completion, for all tasks sharing the same resource. Response time analyses are available from real-time research for a large variety of different schedul-

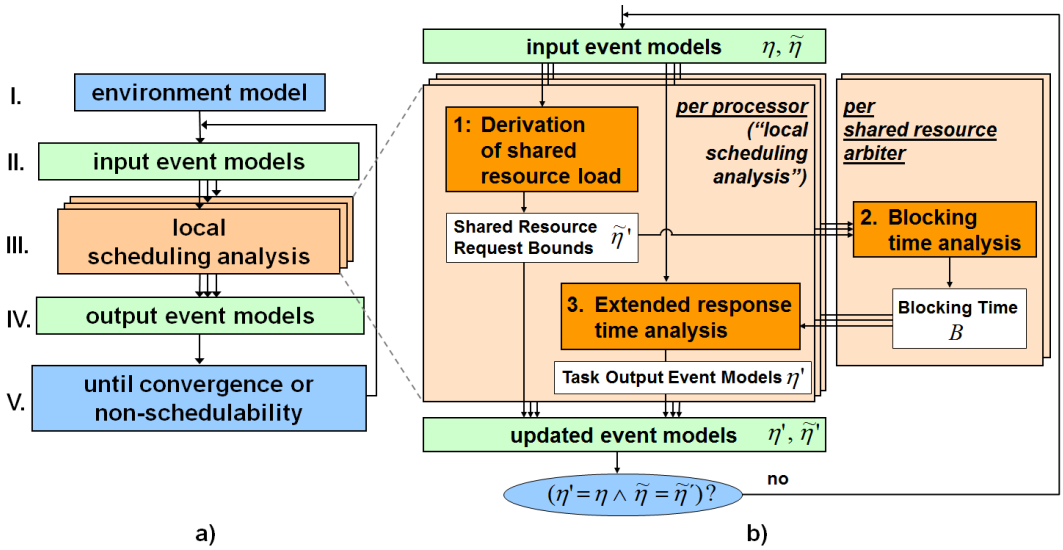


Figure 2.5: a) Classic CPA procedure; b) Extended CPA procedure for multi-core systems with shared resources.

ing policies (SPP, SPNP, RR), which can be directly applied. Many of these analyses are based on the *busy window technique* proposed by Lehoczky [78] and extended by Tindell [154].

Whereas traditional local scheduling analysis only consider independent resources (e.g. busses or processors with different tasks), multi-core systems include also tasks which require access to other shared resources (e.g. memory controllers or a mutex variable) during their execution. Therefore, the local analysis procedure in the classic CPA procedure has to be extended for analyzing the timing behavior of platforms which accommodate secondary shared resources. Figure 2.5b) illustrates the methodology proposed in [90, 130] to capture the inter-core timing dependencies and calculate bounds on the tasks' response times even in the presence of dynamic scheduling and shared resources.

The main idea of this extension is to separate the components' local timing analysis procedure into three disjoint steps [90, 130]:

1. First, the shared resource request bound i.e. the load imposed by tasks on shared resources $\tilde{\eta}$ has to be determined. As shown in Figure 2.2 this can be done by considering the pattern of task activations η. In case more detailed information is available, such as the distance between requests issued by each task [129, 3], the overall load imposed on the shared resource can be derived more exactly for each task and all tasks on a processor [134].
2. Second, the information about the load imposed on the shared resources has to be used to derive the maximum delay (e.g. blocking time) that a task may experience when accessing the shared resources. Concurrent accesses to the shared resources are usually arbitrated by an arbitration protocol (e.g. MPCP [116]) similar to the

scheduling policy of the computational resources. Based on the shared resource loads and the specification of the arbitration strategies, a dedicated blocking time analysis (e.g. as in [90]) computes specific blocking times. These blocking times represent input values for the components' local analyses.

3. In a third step the blocking times provided by the blocking time analysis become part of the response time analysis procedure of each task on each component.

The extended local analysis procedure, including the above mentioned three steps, is the focus of Chapter 3. There, different response-time and blocking time analysis equations are provided which cover multiple combinations of real-time specific processor scheduling policies and shared resource arbitration strategies.

IV. To enable the compositional analysis procedure the event models at the output of each component has to be derived. Similar to the tasks' input timing behavior, also the output timing behavior is captured by event models that are determined using the results of the local response time analysis. The common assumption is that tasks produce one event per output for each activating event. The distance between events at the output of a task is mainly a function of the distance between events at the input of the task (i.e. $\delta_{in}^{-}(n)$ and $\delta_{in}^{+}(n)$) and the task's response time jitter. This means that considering that an event of a task suffers the WCRT R^{max} and all following events that arrived within the minimum possible distance are processed within the BCRT R^{min} the minimum distance between any n at the output of the task is given by the response time jitter [121]:

$$J^{resp} = R^{max} - R^{min} \tag{2.5}$$

The output event model of task with an input event model given by $\delta_{in}^{-}(n)$ and $\delta_{in}^{+}(n)$ can be derived with:

$$\delta_{out}^{-}(n) = \max\{\delta_{in}^{-}(n) - J^{resp}, \delta_{out}^{-}(n-1) + d^{min}\} \tag{2.6}$$

$$\delta_{out}^{+}(n) = \delta_{in}^{+}(n) + J^{resp} \tag{2.7}$$

where d^{min} represents a minimum distance which may separate events at the output of a task [136, 130].

The output event models are then propagated to the inputs of the connected components or to the environment.

V. The compositional behavior of the analysis procedure is finally achieved by the connection of the component's inputs and outputs by the stream representations of their communication behavior using event models [58, 32, 119]. The system-level performance analysis is performed by iteratively alternating local (i.e. component) analysis which includes the additional steps for calculating the blocking times due to shared resources, and the event stream propagation between components. During each analysis iteration, the derived output event models η and the load imposed on the secondary resources $\tilde{\eta}$, are compared to those obtained in the previous analysis iteration. If the output event models are the same, the analysis has converged, otherwise the output event models are used as input event models for a new iteration i.e. the local components' analyses are repeated with the refined inputs.

This iterative analysis procedure — alternating local analyses based on current event models and the derivation of updated output event models as shown in Figure 2.5a) and b) — represents a fixed-point problem. For systems containing cyclic dependencies between two or more components, initial event models are required to begin the local analysis. To solve this problem [119, 121] proposed a solution called *starting point generation*. This means that, for each task on each computational or communication resource an *analysis starting point* is generated by propagating the initial event models (i.e. from the environmental model) along all paths of the task graphs. These initial input event models are then used by the local analyses in the first iteration. After the components' local analyses, output event models are derived as discussed above. These will be the new input event models of the second iteration. The iteration continues until (i) a fixed-point is reached i.e. until all task activating event models η and δ and all shared resource request bounds $\tilde{\eta}$ in two consecutive iterations remain unchanged or (ii) an abort condition is reached (e.g. violation of a timing constraint) in which case the system cannot be deemed schedulable.

The problem of reaching a fixed-point in the compositional system-level analysis procedure and its validity were formally addressed in [143, 142] in case the task activating event models are alone subject of the iterative refinement and in [130, 132] in case the shared resource load in multi-core setups have to be additionally considered. Section 2.3.2 will revisit key aspects for solving the fixed-point problem in general, specific details of the new analysis solutions contributed by this thesis for multi-core and multi-mode systems with shared resources being considered later in Chapter 3 and 4.

2.3.2 Solving the System-Level Iterative Analysis Procedure

In order to apply the CPA procedure for obtaining safe upper bounds on the timing behavior of real-time applications, one must (1) prove that a fixed-point solution of the analysis function exists and (2) if a fixed-point exists one must find it.

Relaying on mathematical tools provided by fixed-point theory (i.e. definitions and theorems introduced by Tarksi and Kleene [148]) [143, 132] and [142] addressed the problem of finding a fixed-point of the CPA.

Tarski's fixed-point theorem [148] states and proofs that any order preserving function (Definition 2.5) defined on a complete partially ordered set (Definition 2.6) has at least one fixed-point. Kleene relied on Tarski's theorem and showed that if a fixed-point exists this can be obtained by iteration.

Definition 2.5 (Order Preserving Function) *A function $f : S \to S$ is order preserving if the application of the function f for any two comparable and ordered elements x and y of the set S results in an identical order of the corresponding results, i.e.*

$$\forall x, y \in S : \qquad x \leq y \Rightarrow f(x) \leq f(y)$$

Definition 2.6 (Complete Partial Order - CPO) *A complete partial order exists for a set S if for the tuple $(S, \leq)$, of the set S and of the partial order relation $\leq$ defined*

on S, it holds the additional property that a least and greatest element w.r.t. the partial order exists in S, i.e.:

$$\exists\ min \in S\ \ and\ \ max \in S \mid \forall x \in S:\ \ min \leq x \leq max \tag{2.8}$$

The CPA is an iterative procedure where in each step local analyses are applied to the individual components (i.e. response time analyses based on busy window approach for cores and busses) based on input event models and where output event models are further derived and propagated for a successive analysis step. Thus, we have to deal with a *global analysis function* (Definition 2.9) that iteratively triggers multiple components' *local analysis functions* (Definition 2.8) applied to different *analysis states* (Definition 2.7).

Definition 2.7 (Analysis State as) *An analysis state as_j ($j \in \mathbb{N}$) consists of the parametrization PR_j of the event models EM_i [4] associated to all tasks $\tau_i \in \mathcal{T}$ in the system model $\mathcal{SM}$ such that: $\forall$ tasks $\tau_i \in \mathcal{T}$ ($i = 1..n$) and $j, n \in \mathbb{N}$*

$$as_j = PR_j(EM_1, EM_2, \ldots EM_i, \ldots, EM_n) \tag{2.9}$$

with

$$EM_i = \{\eta_i^+, \eta_i^-, \delta_i^+, \delta_i^+, \tilde{\eta}_i^+\} \tag{2.10}$$

From this definition we implicitly have as_j^i as the analysis state as_j of task τ_i.

From Section 2.2.1 we know that computational and communication tasks are mapped on computational or communication platform elements $PE \in PE_{comp} \bigcup PE_{comm}$.

Definition 2.8 (Local Analysis Function - LAF) *A local analysis function LAF_k applied to a component $k \in PE_k$ maps an input analysis state as_j^i to an output analysis state as_{j+1}^i for each task $\tau_i \in \mathcal{T}$ mapped on that component.*

$\forall PE_k \in PE_{comp} \bigcup PE_{comm}$ and $\tau_i \in \mathcal{T}$, τ_i mapped on PE_k

$$\begin{aligned} LAF_k &: AS_j \rightarrow AS_{j+1} \\ as_{j+1}^i &= LAF_k(as_j^i) \end{aligned} \tag{2.11}$$

Definition 2.9 (Global Analysis Function - GAF) *A global analysis function GAF maps an analysis state as_j to an analysis state as_{j+1} by applying the local analysis functions LAF_k on each platform element PE_k ($k \in \mathbb{N}$) in the system model $\mathcal{SM}$.*

$$GAF : AS_j \rightarrow AS_{j+1}$$

$$as_{j+1} = GAF(as_j) \tag{2.12}$$

where $\forall PE_k \in PE_{comp} \bigcup PE_{comm}$ and $\tau_i \in \mathcal{T}$, τ_i mapped on PE_k

$$GAF(as_j) = f(LAF_k(as_j^i)) \tag{2.13}$$

[4] i.e. input event models and shared resource requests bounds for all task mapped on all computational and communication platform elements of the system model $\mathcal{SM}$ (see Section 2.2.1 and Definition 2.1).

Thus, the global analysis function is in fact a function of repeated application of all local analysis functions which transform the analysis states.

By applying elements of the fixed-point theory by Tarksi and Kleene [148]) on the CPA specific analysis functions defined above, [143, 132] and [142] have formulated the following conditions to find a fixed-point for the compositional system-level analysis procedure (see also Corollary 2.5. in [132] and Corollary 2.13 in [142]):

Corollary 2.1 (Conditions for Convergence of the GAF of the CPA) *The iterative application of the global analysis function GAF in Definition 2.12 converges towards a fixed-point, if*

- *the global analysis function is order preserving with respect to the analysis states*
- *for the set of the analysis states there is a complete partial order (CPO)*

As the global analysis function repeatedly invoices multiple local analysis functions until a general convergence, the problem of finding a fixed-point breaks down to each local analysis function. This means that for each local analysis function the same two conditions above apply [143, 132, 142].

Corollary 2.2 (Convergence conditions on the local analysis functions) *The iterative application of the global analysis function GAF in Definition 2.12 converges towards a fixed-point, if*

- *each local analysis function is order preserving with respect to the input parameters*
- *the set of each system parameter used by the local analysis functions forms a complete partial order*

In other words, the global analysis function is order preserving if all local analysis functions are order preserving and the set of all analysis states forms a complete partial order if all sets of the system parameters form a complete partial order. As shown in [132, 142] it is important that all analysis modules comply with these conditions. For the extended analysis procedure illustrated in Figure 2.5b) this means that the local scheduling analysis per core and all its components, i.e. the derivation of the shared resource load, the extended response time analysis and all parameters used for analysis, fulfill Corollary 2.2. The same must hold for the blocking time analysis which is also part of the overall system-wide analysis. For the purpose of this thesis, the conditions of Corollary 2.2 will be investigated in Chapter 3 and 4 after introducing the new analysis elements for multi-core and multi-mode systems with shared resources.

A key aspect of the iterative CPA procedure is the speed of convergence. This is an important aspect for the practical use of the CPA because it is not enough to know that a fixed-point will be eventually reached (possibly in infinite amount of time) but more important is to know that the analysis will terminate in reasonable amount of time. [143, 142] elaborated on this and showed that the number of analysis steps required for CPA to reach a fixed-point is finite. More exactly, it was shown that in the context

of CPA the set of investigated parameters is finite and therewith also the number of iterations executed by the analysis until convergence or until a constraint is violated on at least one component (see point V in Section 2.3.1).

2.4 Summary and Overview

In this chapter three distinguishable approaches for system-level performance and timing analysis were summarized, namely: simulation, timed-automata and classical real-time scheduling theory with the holistic and compositional system-level extensions. As all three types of system-level analysis procedures share a common underlying way of modeling systems and capturing their properties, a general modeling approach for distributed and multi-core real-time systems was introduced along with the corresponding terminology. For the scope of this thesis the general compositional system-level performance analysis procedure and its dedicated extension for multi-core systems with shared resources has been detailed. The modeling and the analysis approach represent the foundations for the new solutions contributed in Chapter 3 for the analysis of multi-core real-time applications. However, the system model and the analysis methods discussed so far assume only a static set of tasks over time so that they are not directly applicable to multi-mode systems. As the formal performance analysis of multi-mode applications is the other main goal of this thesis, Chapter 4 will extend the system model and the compositional analysis procedure discussed in this chapter in order to contribute new analysis solutions for multi-mode systems.

Before moving into details, remember that the main goal of the verification steps in the development process of embedded real-time systems is to confirm the adherence of the systems' functional and non-functional/timing behavior to the specified functional and timing requirements. In other words, the verification procedures applied offline must guarantee that at runtime a system will always behave (i.e. under any system internal or external circumstances) according to the specifications. With respect to timing, this means that real-time system designers must guarantee in advance that the systems' timing constraints will never be violated during operation.

3 Timing Analysis of Multi-Core Systems with Shared Resources

3.1 Introduction

Driven by the increasing demand for computational power and by the rising applications' complexity in various embedded application domains, multi-core architectures emerge as the prevalent platform for embedded real-time applications. As highlighted in Chapter 1 strong trends towards multi-core architectures can be observed in communication, media-processing and, more recently, in automotive applications, where multi-core processors are provided by the semiconductor industry (e.g by Freescale [50, 52, 53] and Infineon [66, 67]) and the AUTOSAR standard introduced support for partitioned multi-core OS [12].

The new multi-core processors are aimed to host the significant increase in the computational workload of next generations embedded real-time applications on as few processors as possible. In the automotive domain, this move, co-enabled by the AUTOSAR software interface standards, aims to improve function integration, save costs, and improve maintainability. By using powerful multi-core processors, it will be possible to integrate the functionality of several ECUs (Electronic Control Units) into a single chip or to parallelize complex computations over multiple cores, e.g. in relatively high-performance domains such as engine control or advanced driver assistance systems, in order to allow their extension with modern features that would not be possible without the additional computational power.

While the implementation of multi-core solutions generally delivers additional performance more cost efficiently, their application also introduces a new level of inter-core dependencies that was not previously observed in distributed (automotive) systems. The use of physically shared hardware (e.g. shared memories, I/O devices, coprocessors) and synchronization via logical resources (semaphores) introduces dependencies between task executions on different cores, thus challenging the real-time behavior of the entire system [90, 131, 130, 93]. The application of such multi-core components in safety-critical real-time systems requires careful investigation of the implications on system timing. Consequently the availability of appropriate analysis methods for the prediction of the timing behavior is essential for the design of reliable multi-core real-time systems.

To provide the necessary timing guarantees for multi-core systems, various formal scheduling analysis techniques have been proposed for covering partitioned and non-partitioned multiprocessor scheduling with varying degrees of generality. However, most known schedulability tests are constrained to setups with periodic or sporadic task activation pattern, with deadlines no larger than the period, or no support for shared

resource arbitration, which is frequently required for embedded real-time systems. The current practice requires support for realistic system configurations that exhibit non-periodic task activations, event-driven task activations between dependent tasks, and arbitrary task deadlines.

Furthermore, a critical aspect of efficient system design is the accuracy of the available analyses. As some of the existing methods provide only inaccurate upper bounds on the derived blocking times, new analyses are required to provide more accurate but still conservative results.

Also, most of the proposed approaches for multi-core systems consider the problem of preemptive scheduling, while non-preemptive schedulers in multi-core systems received less attention. Even more, while there are some techniques proposed for the scheduling analysis of tasks which share resources in preemptively scheduled multi-core systems, there is no scheduling analysis solution available for multi-core systems in which tasks that share resources are non-preemptively scheduled.

There are two main reasons why this subject needs more attention. Firstly, non-preemptively scheduled systems are widely used in current real-life applications. For example, most current automotive applications are arbitrated with real-time capable operating systems based on the OSEK/VDX specification [100], which defines priority based preemptive and non-preemptive scheduling and allows resource synchronization via locks administered according to the Priority Ceiling Protocol (PCP) [100, 116]. The scheduling techniques and the synchronization mechanism from OSEK/VDX have been inherited by the most recent AUTOSAR OS and multi-core OS specifications [12]. Secondly, the current evident evolution of automotive ECUs (Electronic Control Units) towards multi-core architectures will be made by maintaining, as much as possible, the backward compatibility with the current solutions. Thus, non-preemptive tasks as defined by the OSEK/VDX standard will be mapped on multi-core processors possibly sharing common resources with other tasks mapped on the same or on other cores [12]. This is a crucial aspect, as non-preemptive scheduling in single-core processors avoids the synchronization overhead due to resource sharing mechanisms and therefore was not considered as a problem.

Besides the fact that non-preemptive scheduling was not considered before in the multi-core context, the more complex setup consisting of the combination of preemptive and non-preemptive scheduling was neglected so far. Nevertheless, as already mentioned, this combination is of particular relevance for the next generation of AUTOSAR conform automotive multi-core ECUs where preemptive and non-preemptive scheduling will coexist on each processor core.

All the aforementioned limitations are handled in the rest of this chapter and overcome through the contributed analysis solutions. In what follows, Section 3.2 presents more exactly the capabilities of the approaches provided by related work and highlights the need for the new analysis solutions contributed by this thesis. Section 3.3 introduces the system model used by the analysis approaches for multi-core systems with shared resources. Further, Section 3.4 highlights key components of a safe synchronization

protocol for shared resources in multi-core systems and discusses the impact of different design decisions with respect to shared resource arbitration and multi-core scheduling policies on the timing behavior of multi-core systems. Sections 3.5 to 3.9 introduce novel blocking-time and response-time analysis solutions for hard real-time multi-core setups under different scheduling policies and shared resource arbitration strategies. The integration of the new analysis equations in the compositional system-level analysis procedure discussed in Section 2.3 is addressed in Section 3.10. Dedicated experiments, introduced in Section 3.11, underline the timing implication of sharing resources in multi-core systems and the applicability of the developed approaches to next generation multi-core controllers, especially for those dedicated to automotive applications.

3.2 Related Work

This section surveys related work in the field of formal performance analysis for multi-core systems, the focus being on multi-core scheduling policies and the corresponding schedulability tests and on synchronization mechanisms for shared resources.

Scheduling policies and resource sharing protocols for multiprocessor and multi-core systems have received significant attention in the last years, however these two topics have not always been jointly addressed. In multiprocessor scheduling, shared resources are often not considered, since traditional multiprocessors used (mostly) local resources. However, for the purpose of this thesis, we are mainly interested in approaches that handle the timing of multiprocessor and multi-core systems with shared resources. Therefore, we will discuss related work on multiprocessor scheduling in general but, insist more on those approaches considering also real-time locking protocols.

3.2.1 Multiprocessor Scheduling

In literature, multiprocessor scheduling policies are generally classified into two major classes, depending on the way task sets are scheduled: the *partitioned* or the *global / non-partitioned*. More exactly, this classification depends on task priorities and on the task to core allocation that can be made. A categorization of multiprocessor scheduling algorithms depending on these criteria was proposed in [31].

Priority assignment. There are three ways how priorities can be assigned to tasks in multiprocessor systems.

1. Task static priorities - A unique priority is associated with each task, and all jobs generated by a task have the priority associated with that task. An example of a scheduling algorithm in this class is the Rate Monotonic Scheduling (RMS) [79].
2. Job-level static priorities - Each job of a task has a single static priority, but different jobs of the same task may have different priorities. The Earliest-Deadline-First (EDF) scheduling [79] uses such a priority assignment.
3. Job-level dynamic priorities - No restrictions are placed on the priorities that may be assigned to tasks or jobs such that a job may have different priorities at different times, as in case of the Least-Laxity-First (LLF) scheduling [87].

Degree of migration allowed. Analogous to the classification of priority schemes the degree of migration in multiprocessor systems is divided into three classes.

1. No migration - Each task is allocated to a processor and no migration is permitted.
2. Restricted migration - Each job must execute entirely upon a single processor, however, different jobs of the same task may execute upon different processors. Thus, the runtime context of each job needs to be maintained upon only one processor, however, the task-level context may be migrated.
3. Full migration - No restrictions are placed upon interprocessor migration, i.e. every job can migrate at every time to another processor. Parallel execution of a job is not permitted.

Thus, multiprocessor scheduling algorithms are referred to as:

1. *Partitioned* in case no migration is permitted;
2. *Global* in case migration is permited;
3. *Hybrid*, in case elements of both, partitioned and global scheduling, are combined.

In literature (e.g. in [41, 21]), multiprocessor systems, on which the different classes of scheduling algorithms are implemented, are classified based on their capabilities in:

1. Homogeneous - the processing cores are identical and thus the rate of execution of any task is the same of any core. Homogeneous multiprocessors are also referred as symmetric multiprocessors (SMP).
2. Uniform heterogeneous - with the exception of the processing speed, all processing cores have identical capabilities (e.g. same co-processors). In this case the tasks' execution rate depends only on the cores' speed, a processor speed of 2 executing each task twice as fast as a processor of speed 1 [41].
3. Fully heterogeneous - the processing cores are different on both, processing speed and capabilities. This means that some tasks are not able to run on any processing core, e.g. in case the core is not enhanced with the required application specific co-processor.

In this thesis, we focus on homogeneous multiprocessor setups consisting on m identical processing cores that are integrated on the same physical chip. In this context the terminology used in literature for multiprocessor scheduling also apply to our multi-core system model.

3.2.1.1 Partitioned Multiprocessor Real-Time Scheduling.

In case of *partitioned multiprocessor scheduling* a set of tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is divided into disjoint subsets $\mathcal{T}^1, \mathcal{T}^2, \ldots \mathcal{T}^m$. Each of these subsets is statically assigned to one of the m ($m \geq 2$) processors of the multiprocessor setup, which are separately scheduled at runtime. The assignment is performed offline, e.g. manually as common in automotive practice, or automatic, e.g. using bin-packing heuristics [35], and cannot be reconfigured at runtime. As a consequence after receiving their local task sets processors are enabled to run scheduling algorithms as they are known from uniprocessor theory, e.g. the fixed-

priority Rate-Monotonic (RM) scheduling [79] or dynamic priority Earliest-Deadline-First (EDF) scheduling [79].

The additional effort of partitioned multiprocessor scheduling algorithms in contrast to uniprocessor scheduling algorithms consists in calculating a partition of tasks and assign the particular subsets to the m processors. However, within the calculation of a correct partition it must be ensured that every task will meet its deadline. Therefore, *partitioned multiprocessor scheduling approaches generally consist of two interconnected steps: (1) Task Partitioning and (2) Schedulability tests.*

There are many partitioning schemes which have been explored for their applicability in the context of multiprocessor scheduling, e.g. [44, 99, 5, 6, 76, 80, 26]. For the purpose of this thesis we won't provide a comprehensive overview of such algorithms, but briefly show their principle. For more details the interested reader is referred to the provided bibliography.

The most partitioning algorithms are based on the RM priority assignment in the context of RM scheduling or on the dynamic priority assignment corresponding to the EDF scheduling and assume tasks with implicit deadlines, i.e. tasks' deadlines are equal to their periods. Multiprocessor partitioning algorithms implement first a bin-packing strategy based on the utilization of tasks, utilizations that are often sorted in a decreasing order. In the second step, schedulability tests, such as the RM bound [79], are applied on each core. By applying these two steps for different combinations of core local scheduling policies, bin-packing heuristics and schedulability tests, research work formulated statements about the maximum utilization bound that can be achieved on a multiprocessor system without shared resources.

For example [6] states that an implicit-deadline task set is schedulable under partitioned RM scheduling on m processors if the task set utilization is up to 50%, i.e $\forall \tau_i$, $\sum U_i \leq m/2$. A similat statement makes [81] which says that an implicit-deadline task set is schedulable under partitioned EDF scheduling on m processors if the task set utilization is $\forall \tau_i$, $\sum U_i \leq (m+1)/2$ when using either the first-fit, best-fit, or worst-fit decreasing heuristic. Arbitrary sporadic task systems on preemptive multiprocessor systems under the partitioned paradigm were investigated in [16]. Similar to the previous approaches, the algorithm in [16] uses an utilization based schedulability test.

Such upper bounds represent a sufficient, but not an exact, schedulability test. Therefore, as also recognized in literature, it is generally preferable to simply partition the task set across the available processors and to apply a response-time analysis to each of them. Since the partitioning of tasks among processors separates the multiprocessor scheduling problem into multiple uniprocessor scheduling problems, well developed analysis techniques, e.g. the response-time analysis based on the busy window technique proposed in [78] and extended in [154] or the load based schedulability test proposed in [79], can be directly applied as long as no secondary resources are shared by the processors. This is also the approach we followed in the papers underlying this thesis, where the mapping of the tasks is assumed given and the focus is on the response-time analysis procedures.

3.2.1.2 Global Multiprocessor Real-Time Scheduling.

Global scheduling algorithms, as opposed to the partitioned scheduling approaches, do not define a static assignment of several tasks to a certain processor. Instead the system behaves dynamically in the sense that instances of the same task, or even different parts of the same instance, can execute on different processors. Thus, in the global scheduling approach, the scheduler maintains a single scheduling queue of ready tasks, from which tasks are dynamically dispatched on the available processors and possibly migrated during execution.

Similar to the partitioned approaches, global multiprocessor scheduling solutions take static and dynamic priority assignments into account. Depending on the priority assignment strategy global multiprocessor scheduling algorithm are usually referred as global FP (fixed-priority), global RM (fixed-priority according to the rate-monotonic policy), global DM (fixed-priority according to deadline-monotonic policy) and global EDF (dynamic priority assignment depending on the earliest-deadline-first strategy).

Global multiprocessor scheduling algorithms based on static priorities work analogous to their counterparts in the uniprocessor world, with the difference that the scheduling algorithm has to select the m (and not only one) highest priority tasks that reside at the RUNNING state. Each of these tasks has to be assigned to exactly one processor. In this context two major challenges arise for global scheduling algorithms:

1. Determining a global priority assignment for the entire task set.
2. Implementing an efficient dispatching mechanism to assign each of the m highest priority tasks to a certain processor.

If the overhead produced by context switches and task or job migration will be neglected, the second topic isn't of great interest. The response time of the tasks won't be influenced by the dispatching mechanism in this case. On the other hand considering the overhead caused by migration and context switching leads to great differences between an arbitrary task-processor assignment and a systematic algorithm. The aim of a dispatching algorithm is the minimization of preemptions and migrations.

Global scheduling of real-time tasks was first considered in [44], which showed that global scheduling schemes based on RM and EDF scheduling are known to suffer from the so-called Dhall effect. That is, in case of global multiprocessor scheduling no minimum utilization can be guaranteed in the sense that there may be task sets with an arbitrary small utilization that are not schedulable with respect to their deadlines. In other words, task sets with a low utilization can be unschedulable regardless of how many processors are available.

To overcome the shortcoming of Dhall's effect, several priority assignments have been proposed to ensure schedulability for general task sets up to a certain bound. For example, [4] defines an utilization threshold depending on the number of processors and divides tasks into heavy-weight and lightweight tasks depending whether their utilization is below or above the assumed threshold. For global FP, the approach in [4] ensures the feasibility of periodic task sets with implicit deadlines with a utilization not more

than 33%, or up to 50% in the special case of harmonic tasks [1]. Other load based or response-time based schedulability tests for task sets with different activation patterns and deadlines under global FP were proposed, a survey of them being given in [40, 41].

Similarly, a significant amount of related work exists on utilization based schedulability tests for global EDF scheduling, again for tasks with different activation patterns and deadlines [41]. Schedulability tests based on response-time analysis were also proposed [18, 60]. In [18] the authors extend the previously known response time analysis technique from the uniprocessor scheduling theory to globally scheduled periodic and constrained sporadic tasks. The approach essentially consists of an iterative computation of an upper bound on the response time of each task, while using the response times of higher-priority tasks to limit the carry-in interference from those tasks. The response-time test in [60] improves the response-time test in [18] and also applies to task sets with arbitrary deadlines.

Pfair Scheduling. A special case of global scheduling is given by the family of pfair (Proportional Fairness) scheduling algorithms [6]. Pfair is based on the quantum-based scheduling, where the scheduler acts only at integer multiples of a scheduling quantum. The key objective of the pfair scheduling algorithms is to execute each task for a proportion of time that is equal to its utilization. Thus, a task τ_i with a worst-case execution time C_i and an activation period T_i will execute for exactly $\frac{C_i}{T_i} \cdot t$ time units in every time interval t. The consequence is that τ_i would execute exactly C_i time units during an interval of the length T_i. Hence all deadlines will be met.

The "pfairness" enables the design of efficient scheduling algorithms at the expense of a significant complexity. The obvious difficulty is to continuously guarantee the equality of execution time proportion and utilization at all times. The processor needs to be divided into infinitely small time slots and has to perform an infinite number of task switches. This is not possible in a real system, irrespective of the fact that the resulting context switch overhead would grow to infinity.

A solution to this problem was proposed in [6]. The authors suggested a combination of priority driven scheduling based on "weight monotonic" priority assignment [2] together with a scheduling policy trying to fulfill the pfair criterion approximately by using a time quanta of size 1. Based on this combination the solution in [6] ensures schedulability for multiprocessor systems with an utilization of up to 50%.

Even if interesting from the scheduling point of view, the overhead involved by the pfair scheduling call its practically into question. For more details on pfair scheduling see e.g. [41] and the references inside.

[1] In a harmonic task set, for all pairs of activation periods (T_i, T_j) either T_i is a multiple of T_j or T_j is a multiple of T_i.

[2] In contrast to other algorithms using static priorities Weight Monotonic Scheduling does not assign priorities with respect to periods or deadlines but corresponding to the utilization of each task.

3.2.1.3 Hybrid approaches

To take advantage of the benefits of partitioned and global multiprocessor scheduling, hybrid approaches were also proposed [30, 74, 21]. More exactly there are two categories of hybrid multiprocessor scheduling approaches namely, semipartitioned and clustered.

The principle of semipartitioned approaches is to split one or more tasks between processors. [74] proposed a semipartitioning method, called PDMS_HPTS (Partitioned Deadline-Monotonic Scheduling under deadline monotonic priority assignments, when used with Highest-Priority Task-Splitting), for sporadic task sets with implicit and constrained deadlines under fixed-priority scheduling. The solution in [74] follows the general partitioned multiprocessor approach, i.e. task are allocated to processors and schedulability test are applied to verify schedulability, with the difference that in case of schedulability fail, the task with the highest priority on each processor is split. With this procedure an utilization bound between 60% and 69% can be ensured.

Clustered approaches combine partitioned and global multiprocessor scheduling in the sense that tasks are partitioned to clusters comprising c ($c <= m$) processors, each cluster being independently scheduled according to a global multiprocessor scheduling policy [30, 21]. In this case, there is no need for specific schedulability tests since already available solutions can be directly applied at cluster level. The case of $c = 1$ corresponds to partitioned scheduling and $c = m$ is equivalent to global scheduling. As a conclusion of [21] the practicality of clustered approaches for hard real-time systems is still limited by the overhead involved by global scheduling.

3.2.1.4 Summary of multiprocessor scheduling.

Partitioned and global multiprocessor scheduling approaches were extensively discussed in literature. Both approaches have their individual drawbacks [18, 21] and thus, no approach has been found to dominate the other over the complete spectrum of possible application scenarios [5, 4, 21].

However, whereas the partitioned paradigm is already adopted in industry specific standards (AUTOSAR [12]), global multiprocessor scheduling in safety-critical (automotive) applications, has not yet found its way into practice. In the near future, this will be difficult because of the high context switching overhead and algorithmic limitations, certification cost in mixed-criticality systems, and not the least, because of the migration cost from legacy industry setups, e.g. OSEK systems in automotive. Hybrid and clustered approaches are promising but for the moment still of limited interest for hard-real time systems where overheads have a significant impact. Therefore, the focus of this thesis is on partitioned multiprocessor setups, where migration is not supported.

Another important observation is that most of the provided analysis solutions for multiprocessor systems consider preemptive scheduling policies in the context of both partitioned and global multiprocessor scheduling. Considerable less attention has been given to non-preemptive scheduling policies. In [15] a test condition was proposed for periodic tasks scheduled non-preemptively according to the multiprocessor global EDF policy. In [62] and [61] the authors have introduced schedulability test conditions for non-

preemptive EDF scheduling and non-preemptive fixed priority scheduling, respectively.

Furthermore, common to all here surveyed approaches is that they don't consider the influence of sharing resources and thus, from now on, of limited interest for this thesis.

3.2.2 Resource Sharing in Multiprocessor Systems

As seen in the previous section, the classical assumption for real-time tasks in multiprocessor scheduling algorithms is that they are independent, which means that tasks do not share any resource beside the processors. In practice however, real-time systems comprise shared resources such as coprocessors, I/O devices, memories, resources which are concurrently used by multiple tasks.

In multiprocessor and multi-core systems, different tasks are mapped on different processors or processor cores which means that shared resources are used by multiple processing elements. In practice, mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by concurrent tasks. The problem of designing such an algorithm is one of the classic problems in concurrent programming.

In the mutual exclusion problem, a task accesses the resource to be managed by executing a critical section of code. Critical sections in multiprocessors can be used to protect either local resources (*local critical section*, *lcs*) shared only by tasks mapped to the same processor / processor core, or global resources (*global critical section*, *gcs*) that are used by tasks on different processors / processor cores. Mutual exclusive access are enforced by hardware arbitration (memory) or are software controlled using, e.g., semaphores or cache based synchronization such as LL/SC as in ARM architectures.

In literature, there are two main categories of synchronization mechanisms mentioned [25]: *non-blocking* synchronization with lock-free execution and wait-free execution and *blocking / lock-based* synchronization. From these two categories, lock-based synchronization techniques are commonly used in practice, including the automotive domain [100, 12]. Considering their practical relevance and the context of this thesis we will further focus only on the related work regarding lock-based techniques.

Lock-based synchronization can be performed either with *suspending*, in which case a task that has to wait for the required resource suspends and the processor becomes available for other work, or with *spinning*, where tasks perform a busy-wait/spin until the lock of the required resource is released; in this time the processor being kept occupied and thus not available for other tasks. In the first case, locks are called *semaphores* and the arbitration protocols *suspension-based*; in the second case locks are called *spinlocks* and the protocols *spinning-based*.

Independent of implementation the main requirement to any lock-based synchronization mechanisms is to ensure predictable and deadlock free mutually exclusive access to shared resources. Directly related to the requirements for predictability and absence of deadlocks is the need, for any synchronization mechanisms, of guaranteeing bounded blocking times for all real-time tasks which at runtime may have to wait for getting access to some shared resources. In addition, being often implemented in the context of priority-based real-time operating systems (e.g. OSEK and AUTOSAR), synchroniza-

tion protocols shall avoid priority inversion and ensure that high-priority tasks (usually highly-relevant and urgent tasks in the system) are not "exceedingly" blocked by critical-sections of lower-priority tasks (usually less-relevant or at least not so urgent tasks). In fact, the blocking of higher-priority tasks caused by critical sections of lower-priority tasks shall be minimized. However, as tasks access shared resources during their execution and as this is controlled by the processor scheduling policy, the functionality of any shared resource synchronization protocol and therewith the fulfillment of the above mentioned requirements is strongly dependent on the scheduling decisions [93, 24] [3].

Therefore, in what follows suspension-based and spinning-based synchronization mechanisms will be addressed in the context of different multiprocessor scheduling strategies. Over the years, several resource sharing protocols and corresponding schedulability tests have been proposed for such task systems.

The first synchronization protocols for shared resources in multiprocessor systems were the Multiprocessor Priority Ceiling Protocol (MPCP) and the Distributed Priority Ceiling Protocol (DPCP) proposed in [117, 115, 116]. These protocols are essentially an extension of the single-processor Priority Ceiling Protocol (PCP) in the sense that they reduce to PCP when used on a single processor. MPCP and DPCP assume that tasks are scheduled according to the partitioned rate monotonic scheduling (RMS) policy and allow to bound the time a task is delayed by other local or remote tasks due to resource contention. Corresponding to MPCP and DPCP the authors proposed a schedulability condition that needs to be checked for each processor core. However, the original method to derive response times for tasks with shared resources arbitrated according to MPCP provides only inaccurate upper bounds on the resulting blocking times. More severely, the analysis is constrained to the case of purely periodic task activations with task deadlines smaller than their periods.

Improved blocking bounds for MPCP, based on response-time analysis [154], were independently proposed in [90, 130] and [75]. Whereas the solution in [75] considers implicit-deadline task sets (i.e. the deadline of each task equals its activation period), the solution in [90, 130] can handle arbitrarily activated tasks and also data-driven activations generated by chained tasks over multiple resources (cores, buses). The solution proposed in [90, 130] is subject of Section 3.7.

[33] proposed a modified resource control protocol that is similar to MPCP but can also be used together with partitioned EDF.

Another shared resource arbitration algorithm for partitioned multiprocessor real-time system is the Multiprocessor Stack Resource Protocol (MSRP) [54], which uses FIFO spinlocks for synchronizing accesses to global shared resources. A comparison of the suspension-based protocol MPCP and of the spinning-based protocol MSRP was conducted in [54]. The results of the performance comparison showed that neither outperforms the other on the complete spectrum of system setups.

Another spinning-based synchronization procedure, but dedicated to global EDF mul-

[3] Key trade-offs in the design of a safe shared resource arbitration protocol for multi-core systems were highlighted in [93]. The impact of different design decisions is subject of Section 3.4.

tiprocessor scheduling, was presented in [43]. More recent work in the field of global multiprocessor scheduling considered also suspension-based protocols. In [45] the Priority Inheritance Protocol (PIP) was considered and the Parallel Priority Ceiling Protocol (PPCP) was presented. An extension of the PIP under global multiprocessor scheduling was proposed in [84] in order to eliminate the negative effects of priority inversions caused by resource-holding lower priority tasks under global multiprocessor PIP.

The Flexible Multiprocessor Locking Protocol (FMLP) was presented in [20]. This can be used under either partitioned or global scheduling, with static or dynamic task priority assignments, and where resources are protected by spin-based or suspension-based locks. An empirical evaluation of the MPCP, DPCP and FMLP synchronization alternatives for multiprocessor systems under different global and partitioned scheduling algorithms was presented in [25, 23, 21]. The results suggest that non-suspending (spin-based) protocols are often a more efficient choice, in particular when critical sections are short and thus the waiting time is less than the cost of blocking and resuming processes.

In [75] the authors propose the Multiprocessor Priority Ceiling Protocol with Virtual Spinning (MPCP-VS). By considering coordinated approaches for task scheduling, allocation and synchronization, the evaluation in [75] indicates that suspension-based protocols (in this paper the classic MPCP with suspension-based blocking) could in fact behave better than spin-based protocols (here MPCP with virtually spinning) under low preemption costs and longer critical sections. As MPCP-VS allows suspensions it cannot be considered a "true" spinning-based protocol such that the results of the conducted comparison should not be considered generally valid.

A more recent evaluation of the suspension-based variants of the MPCP, DPCP and FMPL is presented in [22]. There, improved results (in comparison to [23]) were enabled by an improved analysis (based on linear programming) of the MPCP and DPCP protocols and by the reduced blocking which can be achieved with the FMLP$^+$ protocol (FMLP$^+$ is a refinement of the FMLP for partitioned scheduling).

Finally, solutions were proposed to support temporal isolation between real-time and non-real-time tasks [46] and predictable resource sharing for the component-based design of multiprocessor systems [96]. In order to ensure temporal isolation between hard, soft and non-real-time tasks in symmetric multiprocessor and multi-core systems [46] introduced the Multiprocessor Bandwidth Inheritance Protocol. The protocol combines suspension-based blocking, spinning-based blocking and task migration in order to reduce task waiting times and can be used with global, partitioned or clustered based multiprocessor scheduling [46]. A synchronization protocol dedicated to component-based system is the Multiprocessors Synchronization protocol for real-time Open Systems (MSOS) [96]. It uses semaphores and assumes partitioned fixed-priority multiprocessor scheduling. Under this assumptions, the protocol enables predictable resource sharing among independently developed and provisioned real-time applications mapped on different cores. The experimental evaluation of the MSOS against the MPCP and FMLP protocols shows that the new synchronization protocol enables composability without any significant loss of performance.

While most of the currently available analysis solutions for partitioned multiprocessor setups consider preemptive scheduling policies, considerable less attention was given to non-preemptive scheduling policies. As already mentioned in Section 3.2.1.4 schedulability test conditions for non-preemptive EDF scheduling and non-preemptive fixed-priority scheduling were introduced in [62] and [61], however, without considering shared resources. The response-time analysis solution in [88] handled for the first time a combination of partitioned fixed-priority non-preemptive multiprocessor scheduling and shared resource arbitration. There, the guidelines for sharing resources in partitioned multi-core setups specified by the automotive standard AUTOSAR [12] were considered in the context of non-preemptive scheduling. The contribution of [88] is subject of Section 3.8.

The recent extensions of the AUTOSAR standard specifications for multi-core systems [12] adopted a spinning-based inter-core shared resource synchronization mechanism. However, following the principle "Cooperate on standards, compete on implementation" the AUTOSAR specifications does not contain implementation details regarding the multi-core synchronization protocol. The suitability of various lock types in an AUTOSAR context was studied in [156]. Whereas valid and valuable for general multi-core real-time systems, the results and recommendations of this paper are not directly applicable in the current reality of the automotive domain. Priority-based scheduling and shared resource arbitration are state-of-the-art in current automotive real-time systems, which are developed in a complex distributed process where multiple software functions are implemented indepent of each other and later integrated into one system. In this context, the implementation of FIFO ordered spinlocks, as one locking type mandated in [156], would enable unwanted priority inversion and therewith uncontrollable blocking times of high priority safety-critical functions (usually subject of high quality development and testing activities) by low priority non-safety critical functions (usually subject of less efficient developement and testing activities). The last ones could wrongly behave at runtime and fill in the FIFOs with numerous requests for shared resources, which would delay the requests of the high-priority functions. In this context, imposing a priority-based execution of functions with different criticalities is an important design decision for ensuring the safe execution of the most relevant functions. Furthermore, the study in [156] considers only implicit deadline tasks and partitioned fixed-priority preemptive multiprocessor scheduling.

Thus, despite the mentioned practical relevance, the more complex AUTOSAR conform automotive setup consisting of the combination of preemptive and non-preemptive multiprocessor scheduling for arbitrarily timed and data-driven activated tasks, was not handled so far. The general analysis procedure for AUTOSAR conform multi-core setups is subject of Section 3.9.

3.3 Multi-Core System Model

Relying on the general system model in Section 2.2, this section recapitulates and refines the multi-core system model and summarizes the terminology used by the multi-core timing analysis solutions.

Figure 3.1a) shows a possible evolution of a simplified subsystem, that may be part of a larger system, towards multi-core architecture. For exemplification, consider the initial subsystem composed of four single-core processors and a communication bus. In the multi-core setup one of the single-core processors is replaced by a multi-core processor, for example in order to accommodate further functions or to distribute the load over several cores.

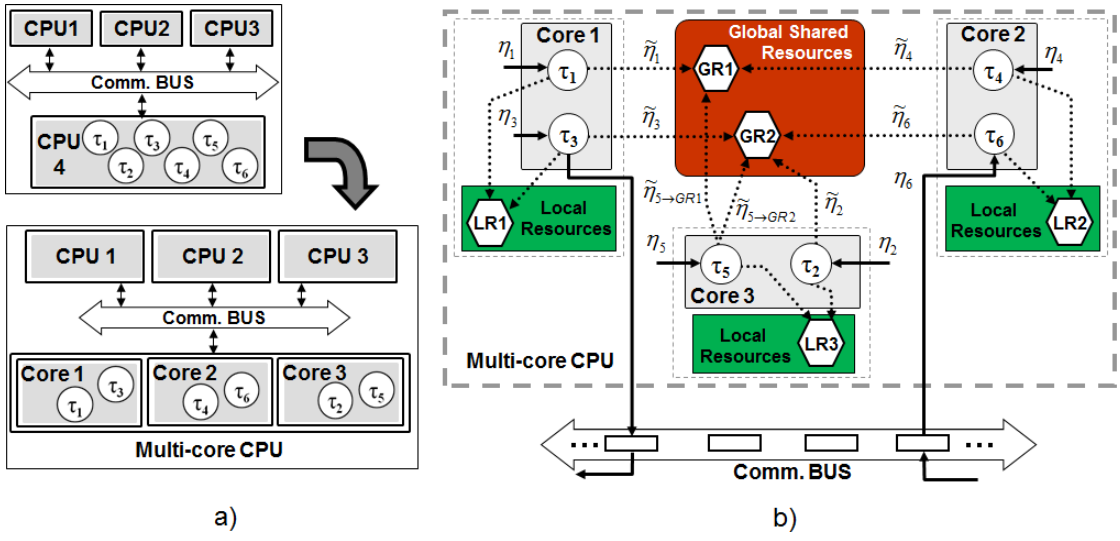


Figure 3.1: a) Example system with three single-core CPUs and one multi-core CPU connected to a communication bus. b) Detailed view of the multi-core CPU with tasks accessing local and global shared resources.

In this chapter, we are particularly interested in the behavior of a multi-core component (as illustrated in Figure 3.1b)), which itself consists of: (i) a set of m ($m \geq 2$) processor cores , each being individually scheduled according to a static priority scheduling policy (e.g. static priority preemptive or static priority non-preemptive), (ii) local shared resources (LR) which are restricted to individual cores, and global shared resources (GR) which can be accessed from each of the m cores and (iii) a static set of arbitrarily activated real-time tasks $\mathcal{T} = \{\tau_1, \ldots \tau_n\}$.

The tasks are considered statically mapped to the available processors with some method e.g manually (as common in automotive practice) or automatic e.g. by using a bin-packing heuristic [76] and our goal is to determine the schedulability of the given mapping. A common priority space is assumed across all m cores and each task in this system has a unique static priority indicated by its index - the lowest index is allocated to the highest priority task. In the system example in Figure 3.1b) task τ_1 has the

highest priority.

Each instance of a task τ_i, called a job and denoted with J_i, is activated by an event, which can be either external (such as interrupts) as in case of tasks $\tau_1, \tau_2, \tau_3, \tau_4$ and τ_5 in Figure 3.1, or the result of another task or bus communication being finished (in which case there is a partial order between the possible task activations) as in case of task τ_6.

Task activation patterns are expressed with event streams (see Figure 2.1 in Section 2.2.2) using the upper event arrival function $\eta_i^+(\Delta t)$, and the lower event arrival function $\eta_i^-(\Delta t)$. These specify the maximum and the minimum number of events that occur in an event stream during any time interval of length Δt. Inversely, event streams can be specified using the functions $\delta_i^+(n)$ and $\delta_i^-(n)$ that represent the largest and smallest time window in which n ($n \geq 2$) events can be observed in the stream. In the system example in Figure 3.1b), the task activating event models are denoted with η_1 to η_6, where the index identifies the activated task.

Each job of a task τ_i is further characterized by its worst-case execution time C_i and its (relative) deadline D_i, which may be smaller, equal, or larger than the distance to the successive activation. Thus, if a task has a worst-case response time larger than this activation distance, it is possible that another instance of this task may be activated before the previous one has completed. In this case, jobs are executed in order, i.e. new jobs may not start execute before the previous job finishes its execution, and this queueing time will be considered as part of the job's response time.

During their executions, jobs of the tasks make use of core local shared resources (LR) and of global shared resources (GR). Accesses to shared resources are arbitrated according to a lock-based arbitration policy, e.g. the Multiprocessor Priority Ceiling Protocol MPCP [116] which specifies rules for accessing local and global shared resources in multi-core setups. Shared resources are assumed to be objects that require serialized access. Jobs address the required shared resources through system calls like GetResource(SRx)/ReleaseResource(SRx), in what follows simply denoted getSRx, with SRx indicating the specific shared resource (e.g. getGR1).

Each access to a shared resource is considered a critical section guarded by a semaphore and protecting the shared resource. A critical section guarded by a semaphore and protecting a global or a local resource is called global critical section (gcs) or local critical section (lcs). The maximum size (duration) of a lcs or of a gcs when accessed by jobs of a task τ_i are denoted ω_i^{LR} or ω_i^{GR}. The maximum number of global critical sections that each job J_i of a task τ_i executes before its completion is n_i^G. With $\tilde{\eta}_{i\rightarrow GRx}^+$ or $\tilde{\eta}_{i\rightarrow LRx}^+$ we denote the load imposed by a job J_i on a global resource GRx or a local resource LRx. We simply use $\tilde{\eta}_i$ instead of $\tilde{\eta}_{i\rightarrow GRx}^+$ where the complete index can be deduced from the context.

Table 3.1 provides a complete overview on the parameters and the terminology used by the analysis methods presented in the next sections.

In this section, no explicit statement was made regarding the scheduling and resource arbitration policies. These will be explicitly indicated in the next sections of this chapter when deriving the corresponding blocking-time and response-time analysis equations.

Parameter	Description
i	Priority of a task; lower values of i indicates higher priority.
τ_i	Task with priority i.
J_i	Job of task τ_i.
C_i	Worst-case execution time of task τ_i; C_i is associated to each job J_i of τ_i.
D_i	Deadline of each job J_i of task τ_i.
$\eta_i^+(\Delta t)$, $\eta_i^-(\Delta t)$	Input/Output event model for task τ_i given by the upper and lower event arrival functions which specify the maximum and the minimum number of events that may occur in an event stream during any time interval of size Δt.
$\delta_i^-(n)$, $\delta_i^+(n)$	Input/Output event model for task τ_i given by the minimum and the maximum event distance functions which specify the minimum and the maximum time intervals during which at least n (n $\geq$ 1) events may occur.
$\tilde{\eta}_i^+(\Delta t)$	*Shared Resource Request Bound* which represents the maximum number of requests that may be issued by a task τ_i to a shared resource within the investigated time interval Δt.
n_i^G	Maximum number of global critical sections that each job J_i of a task τ_i executes before its completion.
ω_i^{LR}, ω_i^{GR}	Maximum duration of a local critical section corresponding to a local resource LR and of a global critical section corresponding to a global resource GR when accessed by jobs of a task τ_i.
${}^c\omega_i^{GR}$	Maximum duration of a specific global critical section c.
${}^{cl}\omega_i^{LR}$	Maximum duration of a specific local critical section l, nested [4] in the global critical section c.
$N_i^{N(c)}$	Number of nested local critical sections entered by a job J_i in the global critical section c, where $c = 1 \ldots n_i^G$. If $n_i^G = 0$ there are no used GRs and if $N_i^{N(c)} = 0$ there are no nested critical sections in the global critical section c.
$lpl(i)$, $hpl(i)$	Sets of tasks mapped on the same core as τ_i which have lower and higher priority than τ_i.
$lpr(i)$, $hpr(i)$	Sets of tasks mapped on remote cores as τ_i which have lower and higher priority than τ_i.
$GS_{i,j}$	Set of global semaphores that will be locked by jobs of both tasks τ_i and τ_j.
$\theta_{i,j}$	Set of tasks which are elements of $lpr(i)$ and access elements of $GS_{i,j}$.
$\Theta_{i,j}$	Set of tasks which are elements of $hpr(i)$ and access elements of $GS_{i,j}$.

Table 3.1: Parameters of the Multi-Core System Model

[4] In case the shared resource arbitration policy allows nested calls for shared resources.

3.4 Impact of Multi-Core Design Decisions

Relying on current automotive practice and on related work of the real-time research community this chapter further highlights the impact of different design decisions with respect to shared resource arbitration and multi-core scheduling policies when moving from single-core processor to partitioned multi-core processor architectures.

For the next explanations we refer to the example system model depicted in Figure 3.1b) in Section 3.3. In the purpose of this section, the three cores, which accommodate the statically mapped hard real-time tasks $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$ and τ_6, are scheduled according to an independent static priority preemptive (SPP) scheduler. During their execution the six tasks make use of local shared resources (LR) according to the Priority Ceiling Protocol (PCP) [116] specified also by the OSEK/VDX standard [100]. Accesses to the global shared resources GR1 and GR2 are arbitrated according to a lock-based arbitration mechanism, the difference between different arbitration decisions being discussed in relation to the cores' local static-priority preemptive scheduling policy.

Timing Implications of Multi-Core Components

In single processor ECUs conflicts due to shared resource usage are arbitrated according to the priority ceiling protocol (PCP) and have only a local impact on the timing of the tasks running on the same ECU. In the case of multi-core systems, the use of physically shared hardware (e.g. shared memory), or synchronization via logical resources (e.g. semaphores) introduces a global effect and generates dependencies between the task execution on the different cores. The local execution of a task on a core is now influenced by the local execution of other tasks on other cores, thus challenging the real-time behavior of the entire multi-core ECU and with this the expected benefits of the multi-core setups.

Key requirements for reliable and predictable timing behavior of multi-core systems with shared resources are: the support for mutual exclusion between tasks on the same and on different cores; the absence of deadlocks; the absence of unbounded blocking, for example caused by priority inversion; the presence of an upper bound for the tasks blocking time; and the minimization of this upper bound. Several key aspects which have to be considered by systems designers in order to fulfill these requirements will be next discussed.

A. Local blocking strategy

According to the specifications of the lock-based arbitration policies, a task τ_i which attempts to lock a shared resource will receive the lock if the resource is not already occupied by another task τ_j (which can be local or remote with τ_i). If the resource is already occupied, the task trying to lock the resource will be blocked and either (i) suspends (allowing other local tasks to run on the host core) or (ii) performs a busy-wait thus keeping the local core occupied until it receives the required resource (in case of spinning-based locking protocols).

As the execution of the tasks and therewith the timing of the requests for shared resources initiated by tasks during their execution depend on the scheduling policy, the

arbitration of conflicting accesses for resources is related to the scheduling decisions. Thus, the local waiting strategy has an impact on the amount of parallel requests from the processor: Allowing other tasks to execute also gives them the opportunity to request a lock. While this may have a beneficial impact on the tasks finishing time, it also introduces problems like additional priority inversion, and increased load on the shared resource (which in multi-core systems also impacts the tasks on the other processors), which need to be considered by the blocking time analysis.

B. Order of granting the locks

In the case of multiple coinciding requests to the shared resources the lock arbitration policy must specify the order in which tasks will access the required resources. A possible solution for granting resources is the first-come-first-served (FCFS) arbitration strategy. This method is simple to implement, but counters the prioritization of tasks on their processors. In the setup represented in Figure 3.2a, where the resources are granted in a FCFS manner, a high priority task (in this case τ_1) may be blocked by all other tasks that are using the same global shared resource. This may lead to unacceptable long blocking times for high priority tasks, in this represented with B_{fcfs}. To avoid such situations in hard real-time systems the order of granting accesses on the resources has to be correlated to the tasks priorities, similarly to the approach specified by the single-processor priority ceiling protocol (PCP). In case of priority based resource arbitration policies (as specified by the MPCP [116]), setup illustrated in Figure 3.2b, task priorities are preserved and the possible blocking for higher priority tasks is reduced (see B_{prior}, where $B_{prior} \leq B_{fcfs}$).

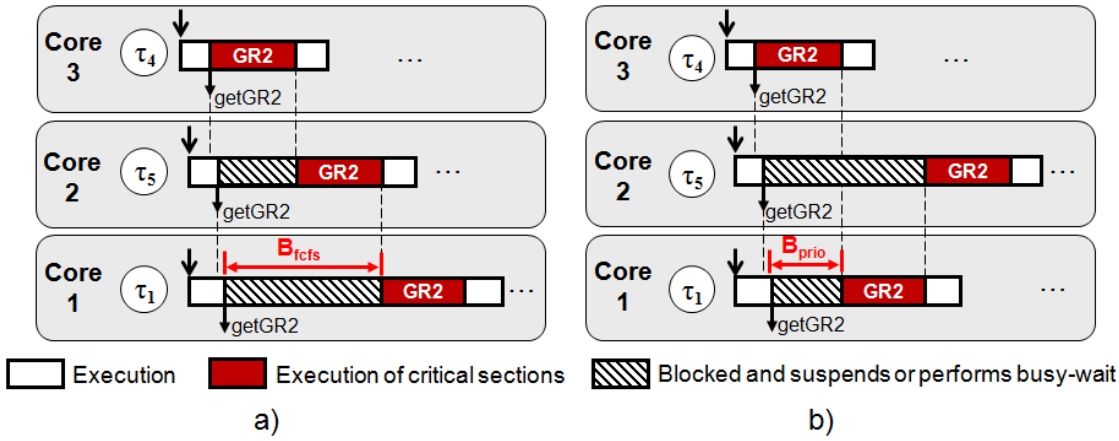


Figure 3.2: Granting resources in a) FCFS manner and b) priority-based manner when tasks on different cores attempt to lock the same global shared resource.

C. Shared resource ceiling priorities

In case of priority based shared resource arbitration, a common approach to avoid deadlocks and unbounded priority inversions between tasks mapped on different cores is to assign ceiling priorities to the accessed shared resources. To highlight the benefit of using priority ceilings in multi-core setups consider the example in Figure 3.3a. Ac-

cording to the static-priority preemptive scheduling approach, the higher priority task τ_2 may preempt the execution of a lower priority local task τ_5 which in the example scenario holds the global shared resource GR2. If task τ_2 will then also try to lock the GR2 it will without further measure have to wait forever for that resource because task τ_5 remains preempted and does not have anymore the chance to release the occupied resource. Such a situation not only influences the core where these tasks are mapped but also the other cores where tasks may also indefinitely wait for the global shared resource GR2 (for example task τ_3 on Core 1 and task τ_6 on Core 2).

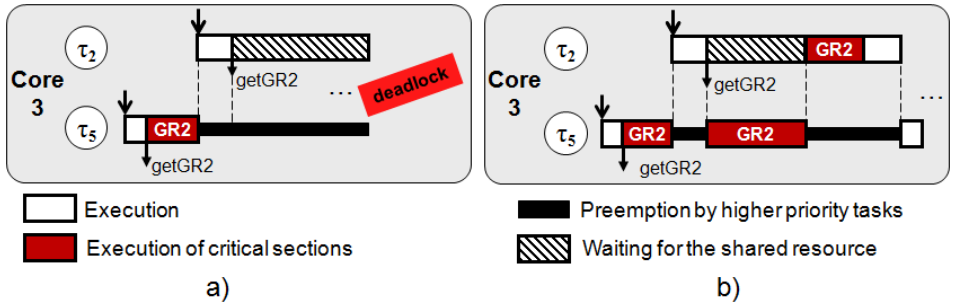


Figure 3.3: a) Deadlock due to waiting for an unreleased global shared resource. b) Using priority ceilings avoids unbounded priority inversion and deadlock situations.

Situations similar to the one presented in Figure 3.3a have been already identified and solved by the priority ceiling protocol (PCP) for single processor systems. The priority ceiling of a local shared resource is defined to be the priority of the highest priority task that may lock that shared resource. A task which locks a shared resource in single-core systems will execute at its assigned execution priority until another task attempts to lock that resource. In this moment the task holding the resource will temporarily raise its execution priority at the priority level of the blocked task and thus will continue executing its critical execution.

A similar approach has to be considered for assigning priority ceilings to global shared resources in multi-core setups (see e.g. MPCP in [116]). Thus, when task τ_2 preempts task τ_5 and attempts to lock the global resource GR2, task τ_5 has to raise its execution priority at the priority level of the global shared resource which has to be higher than the priority of task τ_2. In this way τ_2 will block and the deadlock situation will be avoided (see Figure 3.3b). Because in multi-core systems the blocking occurs among tasks mapped on several cores, it is necessary to assign global shared resources priority ceilings considering the priorities of all the tasks in the multi-core ECU. For this a common priority space across all the cores in the multi-core system must be assumed. As each task on its host processor has a static priority given by its index, the concept can be extended such that each task will have a unique static priority over all cores of the multi-core ECU (e.g. in the system setup in Figure 3.1 tasks have unique static priorities in the range from 1 to 6 with task τ_1 having the highest priority, namely 1). Based on the unique tasks priorities, the priority ceilings of the global shared resources will be assigned such that these will be always higher than the assigned priorities of all

the tasks in the multi-core ECU. In this way a task which locks a global shared resource will temporarily raise its execution priority to the priority ceiling level of the locked resource. This priority assignment strategy ensures that tasks executing global critical sections will not be indefinitely preempted by tasks executing non-critical code and thus will avoid priority inversion and deadlock situations.

D. Preemption of blocked tasks

In order to safely finish the real-time execution of critical tasks, priority-based scheduling policies allow that tasks with higher priority preempt the execution of tasks with lower priority. This raises the question of how to treat preemptions during critical sections or blocking times. In case of multi-core setups the problem becomes more complicated.

As tasks block when requesting for already locked resources the scheduler has to decide what happens during this blocking time. In all the scenarios depicted in Figure 3.4 and Figure 3.5 task τ_5 on Core 3 attempts to lock the global resource GR1, is blocked and has to wait for other tasks on Core 1 and Core 2 that are using and requesting the same global resource. At the moment "A" task τ_5 is still blocked and task τ_2 becomes ready for execution.

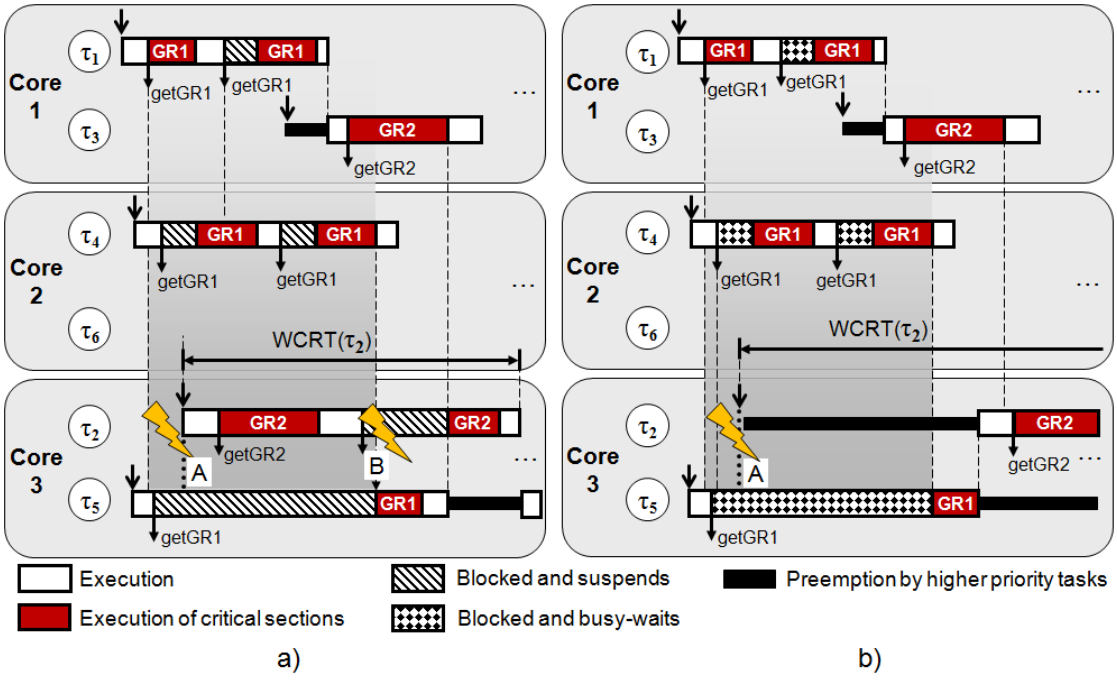


Figure 3.4: Blocking due to global shared resources when a) suspending and b) spinning (busy-waiting).

In case of suspension-based arbitration policies the blocked task will suspend thus allowing other local tasks to execute (see Figure 3.4a the marked moment "A"). In case of spinning-based arbitration when a task is blocked, different decisions are possible,

each of which having its assets and drawbacks for the timing of the other tasks in the multi-core system. A first possible decision is to forbid tasks with any priorities to start executing on their local processor as long as another task is waiting for resources. An example is presented in Figure 3.4b where the blocked task τ_5 does not suspend but performs a busy-wait until it receives the required resource. In this case task τ_2 with higher priority than τ_5 can not start executing on Core 3 until other tasks on other cores will release the resource required by τ_5, and τ_5 will execute the critical code associated to GR1. As such situations may be unacceptable for the real-time behavior of higher priority tasks another possible decision is to allow higher priority tasks to preempt lower priority tasks performing busy-wait (see Figure 3.5 - the marked moments "A"). In comparison to Figure 3.4b, in Figure 3.5 at the marked moments "A" task τ_2 will start executing and during its execution will also queue for its required shared resources. There will be a clear advantage for the timing of the higher priority task τ_2.

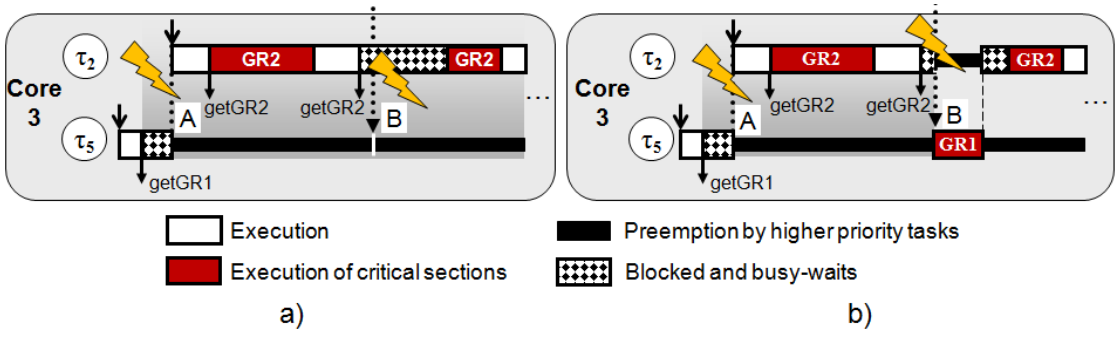


Figure 3.5: a) Preemption of lower priority tasks during busy-wait execution. b) Preemption of higher priority tasks during busy-wait execution when lower priority tasks receive the requested resource.

Further, at the moment "B" in Figure 3.4a and Figure 3.5a and b task τ_2 is blocked (because the required global shared resource GR2 is currently locked) and the resource required by the lower priority task τ_5 becomes available. At this moment, different decisions are again possible.

In the case of suspension-based locking in Figure 3.4a at the marked moment "B" the processor is available for any task ready for execution and τ_5 will execute the critical section associated to the global resource GR1. In case of spinning-based approaches a conceivable decision is to forbid lower priority task to lock the required resource. This case is depicted in Figure 3.5a where τ_5 on Core 3 is not allowed to lock GR1 even if its required resource is available and no other task is executing on the local core. Under this decision the higher priority task τ_2 is privileged and will execute on Core 3 without any further delay when it will obtain the lock of the required resource. On the other hand, if τ_2 will wait for a long time for the global resource GR2 it would be better to allow task τ_5 to execute. Thus, the spinning-based approach may, similarly to the suspension-based approach, also allow the preemption of higher priority tasks when these are performing a busy-wait for a shared resource (see Figure 3.5b). At moment "B" when the global

resource GR1 becomes available task τ_5 can lock GR1, raises its priority at the level of the priority ceiling associated to GR1 and thus preempts task τ_2 when this executes busy-wait. In this case the lower priority task τ_5 is privileged. But, if task τ_5 will execute the critical code associated to GR1 at a higher priority level than task τ_2 when using GR2 and if task τ_5 will lock GR1 for a relative long time the blocking time of the higher priority task τ_2 will significantly increase.

As can be seen, depending on the systems configurations and on the preemption decisions the blocking situations a task will experience on a multi-core system may significantly vary. System designers have to be aware of these details when designing multi-core real-time architectures.

E. Preemption of tasks when executing critical and non-critical code

A further decision which impacts the blocking a task may experience in multi-core setups is related to the preemption of tasks executing critical code associated to a shared resource. A first example of preemption of critical code was already presented in Figure 3.3 where the normal execution of a higher priority task preempts the critical execution of a lower priority local task (in this case task τ_2 preempts task τ_5).

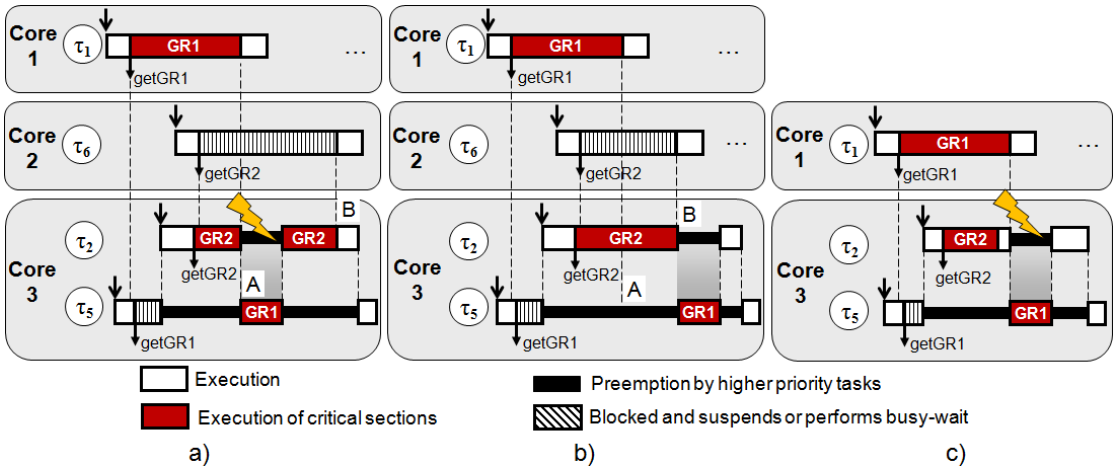


Figure 3.6: a) Preemption of a critical section by other critical section with higher priority. b) Forbid preemption of critical sections. c) Preemption of normal execution by a critical section.

Another possible situation is depicted in Figure 3.6a. The lower priority task τ_5 on Core 3 has an outstanding request for the global shared resource GR1, which has been previously locked by the remote task τ_1 on Core 1. In the meantime, task τ_2 starts executing on Core 3 and locks the global resource GR2. When task τ_1 releases GR1 (the marked moment "A"), task τ_5 may lock this global resource. As τ_5 will raise its execution priority at the level of the priority ceiling of GR1, which is higher than the priority ceiling of GR2, task τ_5 will preempt the critical execution of τ_2. Thus, a task τ_X executing critical code associated to a global shared resource GR_X can preempt another task τ_Y executing critical code associated to another shared resource GR_Y if the assigned

priority of GR_X is greater than that of GR_Y. Note that in case of a multi-core system with more than two cores such a situation may have an additional impact on the other cores. In Figure 3.6a task τ_6 on Core 2 which is waiting for the currently locked global shared resource GR2 will additionally have to wait for the time that task τ_5 preempts task τ_2.

Of course, it is again possible to implement other decisions. For example the protocol could specify that a task with higher priority executing critical code associated to a shared resource (in our example in Figure 3.6b when τ_2 holds GR2 at the marked moment "A") can not be preempted by another local task with a lower priority (task τ_5) even if this would lock a resource with a higher priority ceiling (GR1) than that of the resource locked by the higher priority local task. From the perspective of task τ_2 there is not a significant improvement as task τ_5 executing critical code will anyway preempt task τ_2 after this releases GR2 (see the marked moment "B" in Figure 3.6b). But, there will be an improvement on the timing of task τ_6 on Core 2 which is this case does not have to wait anymore for the critical execution of task τ_5.

Anyway, as the priority ceilings of any global shared resource are higher than the assigned priority of any task in the system, a task executing critical code associated to a global shared resource will preempt the normal execution of another task on its local processor. An example is presented in Figure 3.6c, where task τ_5 within a global critical section associated to the global resource GR1 preempts the normal execution of task τ_2.

Each of these particular scenarios is possible to occur during the execution of tasks in multi-core systems. These examples clearly highlight the dependencies between the tasks execution on the different cores caused by the usage of shared resources in multi-core setups. It depends on the specifications of the resource arbitration policy what types of blocking a tasks running on a multi-core system will experience.

F. Nesting of shared resources

Depending on the applications design, nested calls for local and global resources may be a requirement (e.g. when copying data from one part of the memory to another). But nesting may have severe consequences on the system's reliability due to the excessive resource contention it may generate. A first issue is that nesting easily leads to deadlock situations. An example is presented in Figure 3.7a where two tasks running on distinct cores perform nested calls for the two global shared resources GR1 and GR2. In this case, independent on the locking strategy (suspending or spinning) each task will wait forever for the global resource currently locked and unreleased by the other task. Another example of deadlock is depicted in Figure 3.7b where a task will wait indefinitely for a resource occupied by another local task. Mechanisms to force tasks to release the occupied resources may be implemented but for hard real-time system this may have severe consequences. In order to avoid such situations nesting should be disallowed by construction. Optionally, if nesting is required an explicit partial ordering of calls for shared resources has to be predefined offline.

Beside the deadlock risk, nesting leads to large blocking times for all the tasks in the system. For example, if task τ_5 on Core 3 would be allowed to lock GR1 and GR2, all

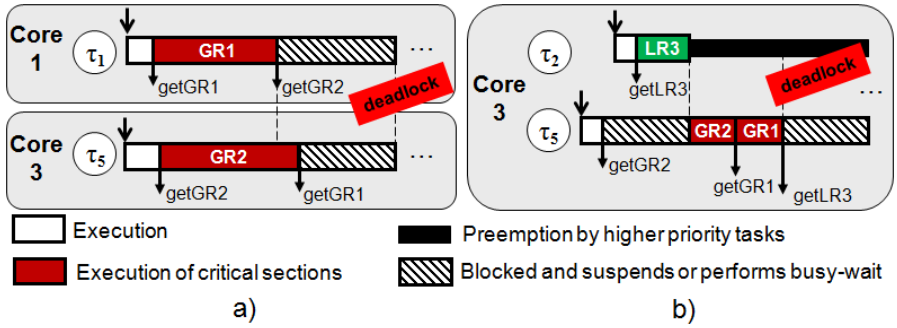


Figure 3.7: Deadlock situations when nesting a) global and b) local shared resources.

other tasks on Core 1 and Core 2 would be blocked if they would request any of the global resources.

The above considered design options clearly show that the blocking types and the associated blocking times of tasks in multi-core setups heavily depends on the specifications of the core local scheduling policies and of the resource arbitration protocols. In order to ensure predictable upper bounds on the tasks timing behavior (i.e blocking and response times) in multi-core setups, the assumed arbitration and scheduling policies have to completely cover all design aspects mentioned above at the points A - F.

The next sections of this chapter show how blocking time and response time analysis equations can capture such design decisions. Based on the introduced timing analysis methods, the impact of different design decisions on the timing behavior of multi-core systems with shared resources will be investigated.

3.5 Principle of the Response-Time Analysis Procedures for Multi-Core Systems with Shared Resources

As stated before, we are mainly interested in the timing behavior of partitioned multi-core setups. Since the partitioning of tasks among processors separates the multi-core processor scheduling problem into multiple uniprocessor scheduling problems, the response-time analysis of partitioned multi-core setups builds up on well developed uniprocessor scheduling techniques. More exactly, we rely on the classical busy window analysis technique proposed by Lehoczky [78] (based on [70]) and extended by Tindell et. al [154] for static-priority (called also fixed-priority) preemptive scheduling (SPP) and Davis et. al [42] for static-priority non-preemptive scheduling (SPNP) [5].

Thus, the response-time analysis approaches for tasks which share resources in partitioned multi-core systems builds up on the response-time analysis of arbitrarily activated tasks in uniprocessor systems.

[5]The busy window concept was also used for the analysis of single-core resources scheduled according to Round-Robin scheduling [114].

3.5.1 Response Time Analysis of Arbitrarily Activated Tasks in Single-Core Processor Systems

Calculation of worst-case response times requires maximum busy window. The worst-case response time of a fixed priority task τ_i on a preemptively or non-preemptively scheduled single-core processor occurs for a job J_i of task τ_i within the maximum priority level-i busy period (called also busy window) [78].

Definition 3.1 *The busy window of a task τ_i on a single-core processor represents a time interval (i) for which the processor executes only tasks of priority greater than or equal to the priority of task τ_i and (ii) during which the processor is never idle [154].*

Derivation of the maximum busy window requires a critical instant. The *maximum level-i busy window* for a task τ_i is built by assuming the occurrence of a so-called *critical instant* [79], where the critical instant depends on the assumed scheduling policy. In case of static-priority preemptive uniprocessor scheduling, the critical instant for a task τ_i is a moment succeeding an idle processor phase when a job of τ_i is activated together with jobs of all higher priority local tasks (i.e. jobs of tasks in $hpl(i)$). In case of static priority non-preemptive uniprocessor scheduling, the critical instant for a task τ_i is a moment just after the job J_j of a task τ_j with the longest core execution time C_j among all local tasks with lower priority than τ_i (i.e. tasks in $lpl(i)$) starts executing after an idle processor phase and where job J_i is released simultaneously with all higher priority local jobs [42]. The maximum level-i busy window ends at the earliest time instant when the processor becomes idle, i.e. when no job of task τ_i or of the higher priority tasks are waiting to be executed.

The critical instant scenarios and the corresponding maximum busy windows in case of static-priority preemptive and static-priority non-preemptive scheduling are illustrated in Figure 3.8 for a task τ_i on a single-core processor, under the assumption that tasks

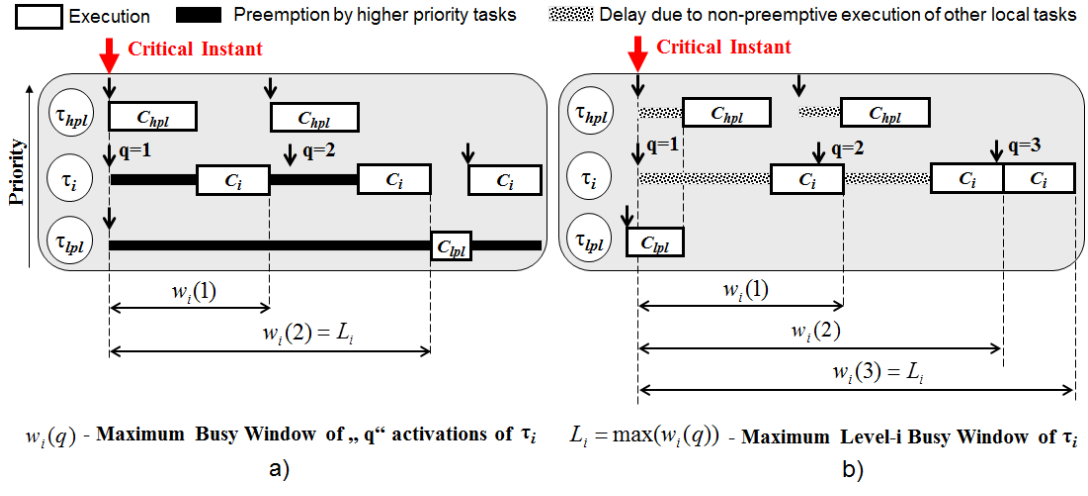


Figure 3.8: Scheduling example and maximum busy windows for a task τ_i on a single-core processor scheduled according to a) SPP and b) SPNP scheduling.

mapped on that processor don't perform accesses for external shared resources.

In case of static-priority preemptive scheduling, the first activation (i.e. $q = 1$) of task τ_i experiences the critical instant when released simultaneously to an activation of task τ_{hpl} and where all subsequent activations of tasks τ_{hpl} and τ_i arrive as early as possible. In case of non-preemptive scheduling, the first activation of task τ_i arrives just after the execution of the first activation of the lower priority task τ_{lpl} started and simultaneously to the activations of the higher priority task τ_{hpl}.

If secondary shared resources are involved, the classical critical instant in case of SPP scheduling must be revisited. Assuming the arbitration of shared resources is performed according to the Immediate Priority Ceiling Protocol (implementation version of the classic PCP [116] [6]) the definition of the critical instant for SPP, discussed above, has to be extended to consider the execution of the longest critical section of a lower priority local task that could be executed when task τ_i becomes ready for execution [116]. Figure 3.9 illustrates such a scenario. Thus, the first activation of task τ_i experiences

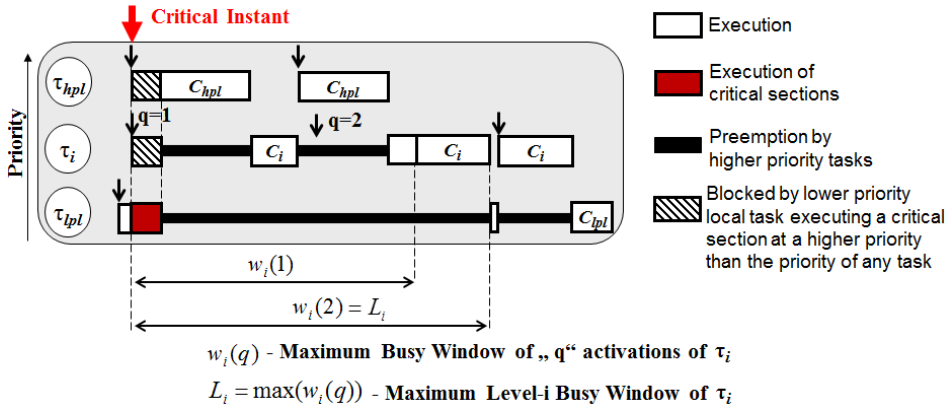


Figure 3.9: Scheduling example and maximum busy windows for a task τ_i on a single-core processor under SPP scheduling and IPCP shared resource arbitration.

the critical instance when released simultaneously to an activation of task τ_{hpl} but just after the lower priority local task has locked a shared resource and raised its priority accordingly. When the lower priority local task releases the shared resource the tasks with the highest priority will start executing.

Note that, in comparison to uniprocessor SPP scheduling, the critical instant under uniprocessor SPNP scheduling does not change when secondary resources are involved. This is because tasks exclusively access shared resources as part of their non-preemptive core local execution.

[6] Priority ceiling based protocols assign statically a priority ceiling to each semaphore such that this is equal to the highest priority task that may use that semaphore. Under IPCP a task which locks a semaphore inherits immediately the priority ceiling of that semaphore. In case of the classic PCP a task which locks a semaphore inherits the ceiling priority only when another task attempts to lock the semaphore. The worst-case blocking time is the same under both implementation variants.

Calculation of the maximum busy window and worst-case response time. The response time of a task τ_i on a single-core processor is given by the largest response time of any of the q ($q = 1, 2, \ldots Q_i$), $Q_i \in \mathbb{N}^+$ task activations (i.e. jobs) that lie within the maximum level-i busy window, denoted L_i .

Assuming the critical instant scenario, the maximum level-i busy window of a task τ_i mapped on a single-core processor with shared resources is classically determined by considering (i) the so called initial blocking time due to the non-preemptive execution in case of SPNP scheduling; (ii) the tasks own execution (i.e. worst-case execution times of its jobs activated in the busy window); (iii) the maximum amount of time the task can be kept from executing due to preemptions by higher priority local tasks, called higher priority interference and (iv) the blocking time in case tasks access secondary shared resources under SPP scheduling. The right hand side of equation (3.1) captures the above four terms, where the blocking time terms mentioned above at (i) and (iv) are captured by a single element BT_i and the tasks' workload is captured by the sum factor [7].

$$L_i^{n+1} = BT_i + \sum_{\forall \tau_j \in hep(i)} \eta_j^+(L_i^n) \cdot C_j \tag{3.1}$$

Thus, BT_i is the blocking due to the lower priority tasks that may execute non preemptively when τ_i becomes ready for execution, $hep(i)$ is the set of tasks with priority higher than or equal to i, i.e. $hep(i) = \tau_i \bigcup hpl(i)$, and $\eta_j^+(L_i^n)$ is the maximum amount of jobs of task τ_j in a time window of size L_i^n and C_j represents their worst-case execution times. The maximum blocking time BT_i caused by the lower priority local tasks is given by:

$$BT_i = \begin{cases} \max\limits_{\forall \tau_j \in lpl(i)} (C_j) & ; \text{ if SPNP scheduling} \\ \max\limits_{\forall \tau_j \in lpl(i)} (\omega_j^{LR}) & ; \text{ if SPP scheduling} \end{cases} \tag{3.2}$$

The clauses in (3.2) capture the blocking time depending on the scheduling policy. Whereas in case of SPNP scheduling the blocking time is given by the lower priority local task with the largest worst-case execution time, in case of SPP scheduling the blocking time is given only by the longest critical section of one of the lower priority local tasks. Remeber that critical sections are modeled as part of the tasks' core execution times.

A solution of equation (3.1) can be computed iteratively, because the right hand side represents a monotonic non-decreasing function which in each iteration either increases by at least C_j or remains unchanged. The recurrence starts with an initial value $L_i^0 = C_i$, and finishes when $L_i^{n+1} = L_i^n$ (i.e. two consecutive iterations provide identical results). The recurrence relation is guaranteed to converge if the resource utilization is less than 100% [42].

The number of task instances that have to be considered when computing the worst-case response time of task τ_i is given by:

$$Q_i = \eta_i^+(L_i) \tag{3.3}$$

[7] Equation (3.1) can be rewritten as: $L_i^{n+1} = BT_i + \eta_i^+(L_i^n) \cdot C_i + \sum\limits_{\forall \tau_j \in hpl(i)} \eta_j^+(L_i^n) \cdot C_j$

To determine the worst-case response time of any task τ_i, it is necessary to calculate the response time for each task instance q ($q = 1 \dots Q_i, Q_i \in \mathbb{N}^+$) within the maximum level-i busy window. The response time of the q-th activation of task τ_i is generally given by the difference between the busy window length $w_i(q)$ and the moment when this activation was initiated relative to the beginning of the busy interval. This is given by $\delta_i^-(q)$, i.e. the minimum distance between q activations, with $\delta^-(1)$ being set to 0.

The equations used for computing the busy windows and the response times of individual task instances differ depending on the core local scheduling policy. Thus, for arbitrarily activated tasks under uniprocessor SPP scheduling we have:

$$R_i = \max_{q=1\dots Q_i} (w_i(q) - \delta_i^-(q)) \tag{3.4}$$

where the maximum busy window $w_i(q)$ of the q-th activation is computed with

$$w_i^{n+1}(q) = q \cdot C_i + BT_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q)) \cdot C_j \tag{3.5}$$

For arbitrarily activated tasks under uniprocessor SPNP scheduling we have:

$$R_i = \max_{q=1\dots Q_i} (w_i(q) + C_i - \delta_i^-(q)) \tag{3.6}$$

where the maximum busy window $w_i(q)$ of the $(q-1)$-th activation (i.e the queueing delay of the q-th activation) is computed with

$$w_i^{n+1}(q) = (q-1) \cdot C_i + BT_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q)) \cdot C_j \tag{3.7}$$

The difference in the equations above (i.e. (3.4) vs. (3.6) and (3.5) vs. (3.7)) is given by the way the execution of the analyzed task instances is considered. In case of SPNP scheduling, the busy window of an instance q of a task τ_i captures separately (i) the queueing delay (i.e. time interval in which the instance q cannot start executing) caused by the execution of the previous $q-1$ instances of τ_i and by the interference the $q-1$ instances of τ_i suffer due to the maximum workload of higher priority local tasks and (ii) the non-preemptive execution of the q-th instance. In comparison to SPNP, in case of SPP the q-th instance is preemptable, fact that leads to the minor differences in the computation procedure.

The recurrence relations (3.5) and (3.7) start e.g. with a value of $w_i^0(q) = BT_i$ and stop when $w_i^{n+1}(q) = w_i^n(q)$, or when the value $w_i^n(q)$ at some iteration point is so large that the obtained response time $R_i(q)$ with (3.4) or (3.6) for the current considered activation q already exceeds τ_i's deadline D_i, in which case the task is unschedulable.

Finally, if worst-case response time values have been obtained for all tasks in the system, the schedulability test consists of checking whether the condition $R_i \leq D_i$ holds for every task τ_i.

3.5.2 Extending Uniprocessor Scheduling Theory

From single-core processors to partitioned multi-core processors. The theory discussed above holds for single-core processors, however, in order to extend it for analyzing multi-core systems with shared resources several aspects have to be addressed. While in uniprocessor systems the blocking time (i.e. the BT term in (3.5) and (3.7)) depends only on tasks mapped on the same core processor, in multi-core processor systems this also depends on the amount of load imposed on the shared resources by tasks mapped on the other cores in the system. Therefore, the derivation of the blocking times of each task τ_i has to take into account system-wide dependencies, dependencies that must be captured during the investigated busy window $w_i(q)$. Furthermore, as shared resource accesses introduce dependencies between the execution of tasks on different cores, the local analysis of one core now depends on the shared resource interference caused by other cores. Therefore, the critical instant scenario and therewith the computation of the maximum level-i busy window $w_i(q)$ have to be revisited in case of multi-core systems with shared resources.

As already mentioned in Section 2.3, three problems have to be addressed in order to calculate response-times of tasks in multi-core systems with shared resources:

1. **Shared resource load derivation**. First, the load imposed by tasks on shared resources has to be determined.
2. **Blocking-time analysis**. Second, this information has to be used to derive the maximum blocking time that a task may experience.
3. **Response-time analyis**. Third, the obtained blocking times need to be integrated in the worst-case response time. This step couples local scheduling analysis with the analysis of the shared resource arbitration, i.e. the blocking time analysis.

As shown earlier in this section the critical instant scenarios and therewith the busy windows depend on the core local scheduling policy. Also, the blocking scenarios and therewith the blocking times of tasks in multi-core setups depends on the employed arbitration decisions (see Section 3.4). Therefore, the critical instance scenarios, the blocking time derivation and the computation of the busy windows and of the response-times, i.e. steps 2 and 3 above, will be addressed in Sections 3.7 to 3.9 for specific processor scheduling policies and shared resource arbitration mechanisms. Common to all multi-core analysis procedures is the derivation of the load imposed by tasks on shared resources, i.e. step 1 above, which is addressed in Section 3.6.

3.6 Derivation of the Shared Resource Load

Lock-based synchronization protocols that ensure mutual exclusion for shared resources accesses are the default choice for guaranteeing safe sharing of data and resources in single-core and multi-core real-time systems. In single-core processor systems, lock-based synchronization mechanisms, as for example the PCP employed by the OSEK OS [100], ensure that a task may be blocked by a lower priority task only once. Such a bound is not so easy to obtain in multi-core processor systems where several tasks execute in

parallel on different cores and through prioritized requests are able to repeatedly lock the required shared resources [90, 93]. In multi-core setups, in a worst-case scenario, each time a task tries to access a shared resource this may be already blocked by another task. Assuming this worst-case scenario for deriving timing bounds is valid but may be really pessimistic. For example, consider the scheduling and resource access example in Figure 3.10 for task τ_1 and τ_4 in the multi-core system in Figure 3.1. Imagine, task τ_1 on

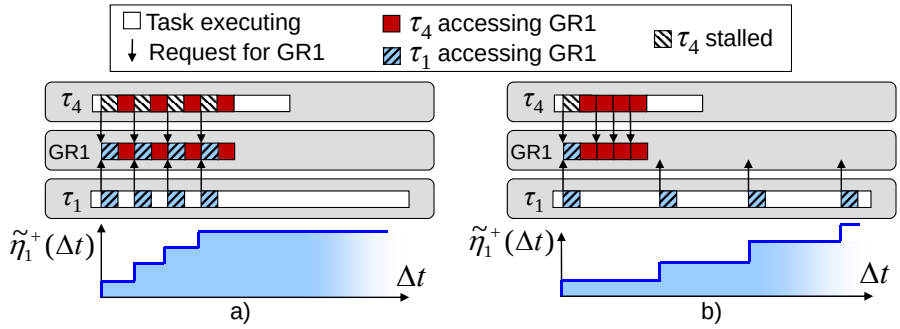


Figure 3.10: Load imposed by task τ_1 on the shared resource GR1.

Core 1 is trying to access the global shared resource $GR1$ that is also used by task τ_4 on Core 2 as depicted in Figure 3.10. The resource arbitration is based on priorities, such that τ_1 receives a higher priority on the resource and thus conflicts are resolved in its favor. Now, assume τ_1 and τ_4 try to access the resource 4 times during their execution. Depending on the timing of tasks' accesses the blocking experienced by task τ_4 differ. If all requests of τ_1 occur at the beginning of its execution (Figure 3.10a), this may cause τ_4 to be blocked each time it tries to access $GR1$. If however, τ_1's requests are further separated in time (Figure 3.10b), during τ_4's execution, only one conflict may actually occur. It is therefore advisable, to take a closer look at how the requests are timed.

The necessity to investigate the timing of the shared resource accesses was identified earlier, load models and the quantification of the dynamic load imposed on the shared resources at runtime being addressed in several publications. Load event models were addressed in [141, 129, 3], these models being later used to derive the runtime load imposed on the shared resources and thereby to perform the analysis of shared resource delays as for example in [135, 134]. Such shared resource delays were also included in the worst-case response time analysis approaches of dynamically scheduled tasks [135, 90, 130, 133, 88], some of these approaches being an important part of this thesis.

The timing of the shared resource load was also addressed in [105, 137]. There, the shared resource load is characterized by event models, however, the analysis approaches assume a constrained preemption model and time-driven superblock scheduling. More exactly, the structures to be scheduled are considered superblocks within which dedicated phases are assigned for local execution and shared resource accesses, the shared resources being arbitrated according to a TDMA schedule.

For the scope of this thesis, we are interested in a general shared resource load model that suits real-life applications, as for example in automotive, where the timing of the

tasks and of their shared resource accesses are highly dynamic and not constrained or isolated by orthogonalization measures as e.g. in [102, 17, 7].

Thus, as shown in Figure 2.2 and Figure 3.10, to capture the shared resource load we rely on the event model concept used to model task activations. The shared resource request bound of a task τ_i, as defined in Definition 2.4 can be straight-forwardly bounded as follows. Let task τ_i be activated by events bounded by event model $\eta_i^+(\Delta t)$, have a worst-case response time R_i, and perform at most n_i accesses to a shared resource per activation. The shared resource request bound $\tilde{\eta}_i^+(\Delta t)$ of such a task in a time interval Δt is then given by:

$$\tilde{\eta}_i^+(\Delta t) = \eta_i^+(\Delta t + R_i) \cdot n_i \tag{3.8}$$

Note that (3.8) features the η_i^+-function shifted by the task's worst-case response time R_i to account for the requests of jobs that are unfinished at the beginning of the investigated time interval. This is required because: (i) the worst-case response time for each task identifies the largest time interval over which the requests of each instance may be distributed and (ii) in multi-core setups, shared resource requests (spread across the response-times) of tasks mapped on different cores may alternate in an unfortunate way and thus maximally block requests of the analyzed tasks.

Regarding the maximum number n_i of shared resource accesses per task instance, this can be obtained by investigating the task's internal control flow. As already mentioned, we assume shared resources which require serialized access and which are explicitly addressed through special instructions in the source code. For example, a task may fetch data each time it executes a for-loop that is repeated several times. By counting the loop iterations per task instance, a bound on the memory accesses can be derived. Focusing on the worst-case execution time problem, previous research provided methods to find the longest execution path and the path with the maximum number of requests (which may not necessarily be the path with the maximum execution time) through a program description [158].

However, depending on the actual system configuration, relying only on the upper bound on the number of requests per task instance may not be sufficiently accurate. In the analysis of the shared resource contention, this may translate into an assumed burst of requests (see Figure 3.10) that may not occur in reality and which finally will result in an overestimated shared resource load. Improved bounds on the resource requests can be derived by measurement, as for example in [135], or, more reliably, by closely investigating the task's internal control flow as in [129] and [3]. The basic assumption of the formal solutions is that for each basic block the execution time is either constant or a minimum execution time and a maximum number of shared resource requests is known. Through program path analysis (i.e. identification of linear execution sequences, jumps, and conditional statements) and knowledge about the task's external activation pattern, distances between multiple requests of a task can be derived. For example, a task that makes an access to a shared resource within a loop, will produce a request sequence that contains several accesses (one per loop) separated by the loop execution time, and the overall pattern repeating with each activation of the task.

If, for any task τ_i a minimum distance d_{srr} between any two shared resource requests is known, the shared resource request bound can be computed for example with:

$$\tilde{\eta}_i^+(\Delta t) = \lceil \Delta t / d_{srr} \rceil \tag{3.9}$$

In comparison to (3.8), the bound calculated with (3.9) has the advantage that it does not require the knowledge of the tasks' worst-case response times, which for some tasks may be unknown at the beginning of the system-level iterative analysis procedure [130]. However, as discussed in Section 2.3 when introducing the compositional system-level performance analysis for multi-core systems with shared resources and the corresponding fixed-point iterative procedure, and as we will see in detail in Section 3.10, the interdependent analysis parameters can be computed through iteration as long as all analysis parameters are monotonic.

Because shared resource accesses are modeled as part of the tasks' core execution times, the distance between two requests can take only certain values. As an example, consider that each job of task τ_1 on Core 1 performs three equally long and non-nestable [8] accesses to the global shared resource $GR1$ (each of size ω_i^{GR1}) during its worst-case execution time C_1 and there is no interference from other tasks in the system. Under these assumptions Figure 3.11a) and b) illustrate the distance between any two requests for $GR1$ when these are as close as possible to each other and as far as possible to each other within the core execution time C_1.

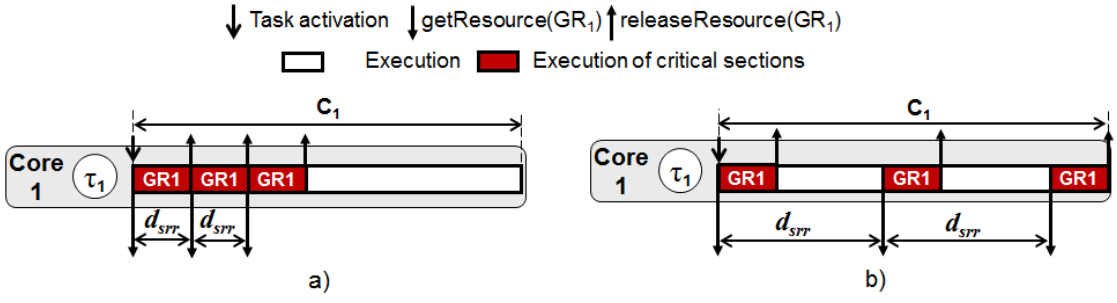


Figure 3.11: Example: minimum and maximum possible distance between two requests for the global resource *GR1* within the core execution time C_1.

In general, depending on the number and the size of the global critical sections per task instance, d_{srr} is delimited as follows:

$\forall$ gcs c among the n_i^G gcs's executed by a job J_i,

$$\min\left({}^c\omega_i^{GR}\right) \leq d_{srr} \leq \frac{C_i - \sum_{c=1}^{n_i^G} {}^c\omega_i^{GR}}{n_i^G - 1} + \min\left({}^c\omega_i^{GR}\right) \tag{3.10}$$

$\min\left({}^c\omega_i^{GR}\right)$ represents the shortest global critical section c among all n_i^G global critical sections executed by any task instance J_i. Note that (3.10) also considers the case where

[8] Note that both, literature [116] and industrial practice [12] recommend avoiding nesting.

global critical sections are of different sizes, in which case assuming always the shortest global critical sections, i.e. $\min({}^{c}\omega_i^{GR})$, is conservative but pessimistic. However, more exact inter-request distance derivation would require more detailed information, i.e. the order of the critical sections within the worst-case execution time C_i.

Information regarding the minimum distance between two requests in one task instance together with information about the tasks scheduling policy and the tasks activating event models (see η^+, δ^- in Section 2.2.2) can be used to reduce the pessimism of the shared resource load bound in (3.8). Further details on deriving $\tilde{\eta}(\Delta t)$ are however beyond the scope of this thesis, but can be found in [134] or in Chapter 5 of [132].

3.7 Response-Time Analysis for Partitioned Static Priority Preemptive Scheduling in Multi-Core Systems with Shared Resources

Based on the shared resource load derivation in Section 3.6, in the following we consider the *Multiprocessor Priority Ceiling Protocol* (MPCP) [116] and introduce an improved blocking time analysis for task sets with arbitrary activation patterns (event models). After that, in Section 3.7.2, the blocking time equations will be integrated in the response-time analysis procedure for partitioned multi-core SPP scheduling.

3.7.1 Blocking Time Analysis for MPCP

Since task deadlines can be larger than their periods, the blocking time analysis has to consider the possible influence of overlapping job execution. This influence can be captured by analysing the execution of tasks during their *busy window*, as discussed in Section 3.5.1 for uniprocessor scheduling (see e.g. (3.5) and (3.7)). The calculation of the maximum busy-window for partitioned SPP multiprocessor scheduling will be introduced in detail in Section 3.7.2. For now assume that we are interested in the blocking time of a task τ_i that accesses local and global resources and is activated q times in a time window of size $w_i(q)$.

3.7.1.1 Specifications of the MPCP

MPCP is a deadlock free protocol which relies on the following assumptions:

- A task τ_i can access local and global resources; a critical section guarded by a semaphore and protecting a global or a local resource is called global critical section (gcs) or local critical section (lcs).
- Priority ceilings are assigned to critical sections;
  - Local critical sections are assigned priority ceilings according to the uniprocessor PCP, thus a local critical section will receive a priority ceiling equal to the highest priority of the tasks accessing the respective local shared resource.
  - Global critical sections are assigned priority ceilings that are higher than the priority of any other task in the system [116], and there exists an ordering of the priority ceilings of the global critical sections.

Assuming P_H to be highest priority of any task in the system, MPCP imposes for each GR a static *base priority ceiling*, denoted with B_{CP}, which is higher than the priority of any task in the system, i.e. $B_{CP} = P_H + 1$. As we assume the task with the highest possible priority to have the lowest possible index we calculate B_{CP} in the negative domain with $B_{CP} = -(P_L + 1)$ where P_L is the task with the lowest possible priority in the system. The *normal execution priority* CP of each *gcs* of a GR^i when accessed by a task τ_i - denoted with $CP(GR^i)$ - is given by: $CP(GR^i) = B_{CP} + \max\{j|\tau_j \text{ uses } GR\}$, where j is the priority of any of the tasks τ_j that access the same global resource GR with τ_i but execute on another core.

- During execution, tasks are suspended when they try to access a locked *gcs*; when a higher priority task is blocked on a global critical section local tasks can be executed and may even try a lock on local or global critical sections.
- Global critical sections are not allowed to be nested in other critical sections (local or global) and vice-versa; if tasks perform nested accesses to global critical sections, an explicit partial ordering of global resources has to be used to prevent deadlocks.

All these specifications make MPCP **deadlock free** and **allow to bound the blocking duration of a job as a function of the duration of critical sections of other jobs and not as a function of the duration of non-critical code**.

3.7.1.2 Derivation of Blocking Times

The blocking time of a task τ_i in a multi-core system with shared resources arbitrated by the MPCP consists of up to five types of blocking. In what follows we extend the five blocking factors of the classical MPCP analysis to consider the influence of multiple job activations and the load imposed on the shared resources. Note that the blocking time equations use the terminology summarized in Table 3.1.

(1) Local blocking time. According to the uniprocessor priority ceiling protocol (PCP), each job J_i of a task τ_i may be blocked once by a job J_j of a lower priority local task $\tau_j \in lpl(i)$. In the occurrence of overlapping activations of task τ_i, a lower priority local job J_j will block only the first job of the task τ_i (once J_j exits the critical section which blocks J_i it cannot execute anymore before all jobs of τ_i are finished).

Additionally, in the multiprocessor protocol, each time a job J_i tries to lock a global semaphore, it can potentially suspend, letting lower priority jobs execute on the local processor. This reproduces the situation presented above where jobs of lower priority tasks can lock local resources each time J_i attempts to enter a global critical section and suspends. These low priority jobs can lock local semaphores and block J_i when it resumes its execution. Therefore, the local blocking time of a job J_i is bounded by:

$$B_{i1}(w_i(q)) = [1 + q \cdot n_i^G] \cdot \max_{\forall \tau_j \in lpl(i)} (\omega_j^{LR}) \tag{3.11}$$

(2) & (3) Direct blocking times. Each time a job J_i tries to enter a global critical section, it can find that this is currently held by a lower priority job on a different

processor. Thus, the blocking time due to lower priority remote tasks which share the same global resources with J_i (jobs of tasks in the set $\theta_{i,j}$) is bounded by:

$$B_{i2}(w_i(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j}} (\omega_j^{GR}) \tag{3.12}$$

Similar, each job J_i can be blocked by higher priority remote jobs that request the same global resource as J_i (jobs of tasks in the set $\Theta_{i,j}$). As opposed to lower priority remote jobs, higher priority remote jobs may be served multiple times.

$$B_{i3}(w_i(q)) = \sum_{\forall \tau_j \in \Theta_{i,j}} (\tilde{\eta}_j^+(w_i(q)) \cdot \omega_j^{GR}) \tag{3.13}$$

(4) Indirect preemption delay. Consider now the processors on which tasks that can directly block task τ_i (tasks in $\theta_{i,j}$ and $\Theta_{i,j}$ [9]) are mapped. Each of these processors may contain other tasks that access global resources with higher priority ceilings than the priority ceiling of the resources accessed by tasks directly blocking τ_i. We denote the set of these tasks with $\Psi_{i,j}$. If tasks on these processors, i.e. tasks in $\Psi_{i,j}$, access global resources with higher priority ceilings than the priority ceilings of the resources accessed by tasks directly blocking τ_i, each of them can preempt the global critical sections of tasks directly blocking τ_i. Their influence on the blocking time can be captured by:

$$B_{i4}(w_i(q)) = \sum_{\forall \tau_j \in \Psi_{i,j}} (\tilde{\eta}_j^+(w_i(q)) \cdot \omega_j^{GR}) \tag{3.14}$$

(5) Local preemption delay. Each time a job J_i of task τ_i tries to access a global resource, it can potentially suspend, letting jobs of lower priority local tasks execute on its local processor. If these jobs require access to global resources (jobs of tasks $\tau_j \in lpl(i)^G$), they can lock or queue up on the global resources and can therefore preempt J_i when it executes non-critical code. Within the investigated time interval $w_i(q)$ there are at most q jobs of task τ_i and each of these jobs can issue maximal n_i^G requests to global resources. In addition, when J_i begins its execution on its local processor, a lower priority job can have an outstanding request for a global semaphore. Hence, in the analyzed time interval, task τ_i can be blocked for at most $q \cdot n_i^G + 1$ global critical sections of tasks in $lpl(i)^G$. But, lower priority local tasks that require access to global resources can issue at most $\tilde{\eta}_j^+(w_i(q))$ requests to global resources within $w_i(q)$. As a result, only the minimum of these two bounds may actually occur.

$$B_{i5}(w_i(q)) = \sum_{\forall \tau_j \in lpl(i)^G} \min(q \cdot n_i^G + 1, \tilde{\eta}_j^+(w_i(q))) \cdot \omega_j^{GR} \tag{3.15}$$

The worst-case blocking time that a task τ_i can encounter in a time window $w_i(q)$ is given by the sum of the five blocking factors B_{i1} to B_{i5} in (3.11) in (3.15).

$$BT_i(w_i(q)) = \sum_{k=1...5} B_{ik}(w_i(q)) \tag{3.16}$$

[9] Jobs of the tasks in $\theta_{i,j}$ and $\Theta_{i,j}$ are jobs which directly block jobs of task τ_i.

For each task τ_i, this blocking time equation is part of the busy-window iterative computation that have to be solved in order to bound the task's worst-case response time under partitioned multi-core SPP scheduling.

3.7.2 Response Time Analysis for Partitioned Multi-Core SPP Scheduling

In this section, we introduce the schedulability condition for arbitrarily activated tasks scheduled according to the partitioned multiprocessor static priority preemptive scheduling and which share resources according to the MPCP arbitration policy. For this, we extend the classical busy window approach introduced in Section 3.5.1 for arbitrarily activated tasks under single-core static priority preemptive scheduling. In a first step, this requires revisiting the critical instant scenario (exemplified in Figure 3.8) in Section 3.5.1) and therewith the computation of the maximum level-i busy window, on which the classical response-time analysis procedure rely.

3.7.2.1 Critical Instant and Maximum Level-i Busy Window for Multi-Core Setups

From the single-core processor theory we know that the worst-case response time of a task τ_i is given by the largest response time of any of the q ($q = 1 \dots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the maximum level-i busy window L_i (see (3.1) and (3.3) in Section 3.5.1):

$$L_i^{n+1} = BT_i + \eta_i^+(L_i^n) \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(L_i^n) \cdot C_j$$

Two important aspects need to be considered in order to extend the busy window analysis equation above for multi-core setups.

Firstly, in case of multi-core setups the blocking time BT_i of a task τ_i is a function of a window size L_i^n during which shared resource requests are issued for local and global shared resources.

Secondly, because of the existing inter-core blocking scenarios one can not rely on the classical critical instance scenario anymore. In [116] it was shown that the use of global shared resources under a suspension-based blocking strategy and rate-monotonic scheduling may lead to deferred tasks' executions, which counters the assumptions regarding the critical instant scenario on which the classical response time analysis approach rely. This means that a job can suspend itself when waiting for a global semaphore to be released and resume and complete its execution to just meet its deadline at the end of the period. In this way higher priority tasks inflict "back-to-back hits" on lower priority tasks [116]. Thus, the busy window of a task τ_i consists not only of the time interval during which task τ_i or a higher priority local task τ_j is continuously executing, but more generally the time interval during which at least one invocation of τ_j is not finished due to remote blocking. This leads to an increased interference for τ_i, which includes also unfinished invocations of τ_j that have started before the investigated busy window. This is covered by shifting τ_j's activation function $\eta_j^+(L_i^n)$ by its worst-case response time R_j.

Thus, the maximum level-i busy window L_i of a task τ_i in partitioned multi-core systems under SPP scheduling can be calculated with the following recurrence relation:

$$L_i^{n+1} = BT_i(L_i^n) + \eta_i^+(L_i^n) \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(L_i^n + R_j) \cdot C_j \tag{3.17}$$

In comparison to (3.1) for single-core processors, equation (3.17) contains new components, i.e. the response time values R_j of the higher priority tasks and the blocking time derivation $BT_i(L_i^n)$, which challenge the classical iterative calculation procedure.

The dependency of the maximum level-i busy window, and therewith of the response time R_i, of a task τ_i on the response times R_j of higher priority local tasks τ_j ($\tau_j \in hpl(i)$) can be tackled by computing all response times and implicitly all busy windows in a top-down fashion, starting with the highest-priority task.

More difficult in the given setup is the fact that blocking factors B_{i3}, B_{i4}, and B_{i5} (in (3.13), (3.14) and (3.15)) and therewith the blocking time $BT_i(L_i^n)$ in (3.17) rely on the resource request bound $\tilde{\eta_j}^+$ in (3.8) and thus indirectly on the response time of potentially *lower* priority remote tasks. This leads to a cyclic dependency. In [116], this problem is tackled with an extension of the resource arbitration protocol by a so called *period enforcer*, which spreads the shared resource accesses over time. The solution we propose in this chapter does not require such modification. The request bound in (3.8) can be computed through iteration as long as all analysis parameters are monotonic or, it can be replaced by the bound in (3.9) that is independent of the tasks' response times.

As presented in [130] and as will be detailed in Section 3.10 all components of equation (3.17) grow monotonically with respect to the window size and therefore allow the iterative calculation of a solution. The recurrence relation (3.17) starts with an initial value $L_i^0 = C_i$, and finishes when $L_i^{n+1} = L_i^n$ (i.e. two consecutive iterations provide identical results).

3.7.2.2 Derivation of the Worst-Case Response Times

Similar to the analysis approach for single-core processors, the number of task instances that have to be considered when computing the worst-case response time of a task τ_i under partitioned multiprocessor SPP scheduling is given by $Q_i = \eta_i^+(L_i)$, with L_i obtained with (3.17) above.

Thus, the worst-case response time of a task τ_i is given by the largest response time of any of the q ($q = 1 \ldots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the busy window $w_i(q)$ as follows:

$$\begin{aligned} R_i &= \max_{q=1\ldots Q_i} r_i(q) \\ r_i(q) &= w_i(q) - \delta_i^-(q) \end{aligned} \tag{3.18}$$

The maximum busy window $w_i(q)$ of the q-th activation is computed with:

$$w_i^{n+1}(q) = q \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q) + R_j) \cdot C_j + BT_i(w_i^n(q)) \tag{3.19}$$

where $q \cdot C_i$ represents the maximum workload of q activations of task τ_i; $hpl(i)$ is the set of local tasks with higher priority than τ_i; $\eta_j^+(w_i^n(q) + R_j)$ is the maximum amount of unfinished jobs of τ_j in a time window of size $w_i^n(q)$; and $BT_i(w_i^n(q))$ is the maximum blocking time computed with (3.16) as presented in the previous section.

Similar to equation (3.17), the recurrence relation (3.19) can be solved by iteration, because all components grow with the window size (for more details and proofs see Section 3.10). The recurrence starts with a value of $w_i^0(q) = q \cdot C_i$ and ends when $w_i^{n+1}(q) = w_i^n(q)$, or when the value $w_i^n(q)$ at some iteration point is so large that the obtained response time for the current considered activation q, i.e $r_i(q) = w_i(q) - \delta_i^-(q)$, already exceeds τ_i's deadline, in which case the task is unschedulable.

Finally, if worst-case response time values R_i have been obtained for all the tasks in the multi-core system, the schedulability test consists of checking whether the condition $R_i \leq D_i$ holds for every task τ_i.

3.8 Response-Time Analysis for Partitioned Static Priority Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources

The new multi-core extensions of the AUTOSAR automotive standard - the dominating automotive software architecture worldwide - uses a combination of partitioned fixed-priority scheduling strategies with preemptive and non-preemptive execution and (potentially) arbitrary deadlines. Since multi-core systems in general use shared resources, this leads to the problem of analyzing preemptive and non-preemptive multiprocessor scheduling with shared resources. While preemptive scheduling has been well investigated in this setup, non-preemptive scheduling analysis is still open and cannot simply be derived. In this section, we address this subject and present an analysis method which allows the calculation of response-times for tasks with arbitrary activations and deadlines which share resources in multi-core systems scheduled according to the partitioned fixed-priority non-preemptive scheduling. Therewith, the contribution of this section provides an essential building block for the analysis of upcoming multi-core real-time applications where both preemptive and non-preemptive scheduling coexist.

This section addresses non-preemptive multi-core scheduling in two steps. Section 3.8.1 addresses the AUTOSAR mechanism [12] (i.e. spinlock-based) for inter-core task synchronization in the context of fixed-priority non-preemptive multi-core scheduling and presents the derivation of the corresponding blocking time bounds. After that, Section 3.8.2 introduces the response-time analysis procedure for tasks with arbitrary activations and deadlines which share resources in multi-core systems scheduled using partitioned fixed-priority non-preemptive scheduling.

3.8.1 Blocking Time Analysis for Multi-Core SPNP Scheduling

It is well known that the overhead due to synchronization mechanisms can be neglected in case of single-processor non-preemptive scheduling. This is ensured by the intrinsic behavior of the non-preemptive scheduling, which in case of single-core processors keeps the execution of the tasks exclusive and therewith also the requests for shared resources. In multi-core setups, this is not the case anymore. Accesses initiated by tasks executing on different cores may interfere as depicted in Figure 3.12 where jobs of the tasks τ_1 and τ_5 in the multi-core system in Figure 3.1b) are blocking each other when requesting the same global shared resource.

Figure 3.12: Conflicting accesses from tasks mapped on different cores.

As there is no synchronization mechanism for shared resources in multi-core systems which explicitly considers the static priority non-preemptive scheduling, we next introduce a dedicated resource arbitration solution. For this, we consider that a spinlock-based mechanism is used exclusively for inter-core synchronization (as proposed in the current AUTOSAR specification [12]) and exploit it in the context of static priority non-preemptive scheduling. This step is needed not only to highlight the impact of sharing resources among cores when assuming non-preemptive scheduling, but also to ensure the premises for deriving bounded task blocking times and therewith a predictable timing behavior of multi-core systems in this setting.

We call the arbitration protocol Multi-core Locking Protocol for Non-Preemptive scheduling and use further the abbreviation MLP-NP.

3.8.1.1 Specification of the MLP-NP Arbitration Policy for Shared Resources in Multi-Core Non-Preemptive Systems

- **Task priorities.** Under static priority non-preemptive scheduling, a job J_i of a task τ_i which starts executing on its host core will run until completion without any preemption by other local tasks independent of the associated priorities. As tasks under non-preemptive scheduling execute exclusively, priority inversion situations or deadlocks can not occur if nested calls for global resources are forbidden or if a global unique ordering is defined (see below). This makes the use of priority ceilings superfluous. Therefore, a job J_i which locks a shared resource (global or local) will execute the associated critical section at the assigned priority.
- **Arbitration of local shared resources.** As a consequence of the local scheduling

policy, during the execution of a job J_i there will be no pending shared resource requests initiated by other local jobs with J_i. Thus, an executing task τ_i will always occupy the requested local resources without any blocking by other tasks.

- **Arbitration of global shared resources.** During execution, a job J_i of a task τ_i which tries to lock a global resource will lock the resource if that is currently not occupied by another remote job. If the requested resource is occupied, the job which has initiated the request will actively wait (spin) until the required resource is released. This means the processor is stalled, and independent of the execution priority no other task mapped on the same core with τ_i is allowed to execute. This is imposed by the local non-preemptive scheduler. An example is depicted in Figure 3.12 where a job of task τ_2, which in this case has higher priority than task τ_5, may not start executing until the active job of task τ_5 completely finishes its execution.

Note that, the AUTOSAR spinlock mechanisms [12] does not explicitly consider the underlying scheduling policy and specify that a higher priority task (e.g. τ_2) could preempt a lower priority task (e.g. τ_5) during spinning. The non-preemptive scheduling counters this assumption and imposes larger blocking times on the higher priority tasks.

In case of coinciding requests for a global resource GR_i, initiated by jobs of different tasks mapped on different cores, the highest priority job requesting GR_i will lock that resource.

- **Nested calls for shared resources.** In order to avoid deadlocks, nested accesses to global resources are not allowed. A job J_i which already holds a global resource is not allowed to request another global resource before the previously locked resource has been released [10]. Nested calls with respect to local resources are permitted. Thus, a job holding one global resource may perform calls for local resources and vice-versa, a job holding one or more local resources may perform a call for one global resource.

The MLP-NP arbitration protocol represents a basic solution proposed with the goal of maintaining the compatibility with the non-preemptive scheduling behavior. In the following, we will derive upper bounds on the blocking times that tasks can experience under MLP-NP. Demonstrating that the blocking time is bounded under all circumstances, we implicitly show that deadlocks are not possible and therewith that the protocol is safe.

[10] Other design decisions could be employed for the nesting of global shared resources and for the arbitration of the global shared resources (e.g. FIFO queues for coinciding requests on global shared resources or *suspension-based* blocking using priority ceilings [116]). If global shared resources shall be nested, an explicit partial ordering of calls for shared resources has to be predefined offline in order to avoid deadlocks and potentially starvation situations (recommended in both, literature [116] and industrial practice [12]). An extended discussion and an evaluation of the trade-offs between the different design decisions regarding the synchronization mechanisms is beyond the scope of this thesis. However, the framework we present in this thesis can be extended to consider other shared resource arbitration schemes, thus making a future comparison possible.

3.8.1.2 Derivation of Blocking Time

Similar to the blocking time analysis for MPCP in Section 3.7.1 the blocking time terms corresponding to the MLP-NP have to capture the overlapping jobs execution during their *busy windows* (see e.g. (3.5) and (3.7) in Section 3.5.1). The calculation of the maximum busy-window for partitioned SPNP multiprocessor scheduling will be introduced in detail in Section 3.8.2. For now assume that we are interested in the blocking time of a task τ_i that accesses local and global resources and is activated q times in a time window of size $w_i(q)$. Note that the blocking time equations introduced next use the parameters summarized in Table 3.1 and the shared resource load derivation in Section 3.6.

The blocking time of a job J_i in a multi-core non-preemptive system consists of the following two blocking factors.

(1) & (2) Direct blocking times. When a job J_i of a task τ_i requests a global shared resource, this can be locked by a lower priority job J_j of a task mapped on a different core than τ_i, i.e. $\tau_j \in lpr(i)$. In the worst-case scenario, each time when J_i attempts to lock a global shared resource, it may find that this is currently locked by another lower priority job on another core (i.e. by one of the jobs J_j of the tasks in $\theta_{i,j}$). Thus, a job J_i is blocked at least for the duration of the longest global critical section ω_j^{GR} as follows:

$$DB_{i,lpr}(w_i(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j}} (\omega_j^{GR})$$

However, if the jobs J_j perform nested calls for local shared resources the maximum sum of nested critical sections (i.e. one global and potentially multiple local) sizes has to be calculated with:

$$S_j = \max_{c=1 \ldots n_j^G} ({}^{c}\omega_j^{GR} + \sum_{l=1}^{N_j^{N(c)}} {}^{cl}\omega_j^{LR}), \forall \tau_j \in \theta_{i,j} \tag{3.20}$$

Thus, the blocking time due to lower priority remote tasks which share the same global resources with J_i can be generally calculated with:

$$DB_{i,lpr}(w_i(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j}} (S_j) \tag{3.21}$$

Similar to the previous blocking factor, each job J_i can be blocked by higher priority remote jobs that request the same global resource as J_i (i.e. jobs of tasks in the set $\Theta_{i,j}$). The largest sum of the durations of the nested critical sections has to be calculated with (3.20). As opposed to lower priority remote jobs, higher priority remote jobs may be served multiple times before the job J_i will be able to lock the requested global shared resource.

$$DB_{i,hpr}(w_i(q)) = \sum_{\forall \tau_j \in \Theta_{i,j}} (\tilde{\eta}_j^+(w_i(q)) \cdot S_j) \tag{3.22}$$

Note that, in case the nesting of global shared resources would be allowed and an explicit partial ordering of the calls for the global shared resources has been defined, the blocking factors B_{i1} and B_{i2} above should consider a sum, similar to (3.20), over all critical sections (global and local) that have been defined as nestable. In this case, the responsibility of the deadlock- and starvation-freedom lies with the offline configured ordering of the calls for shared resources.

The worst-case blocking time $BT_i(w_i(q))$ that a task τ_i can encounter in a time window $w_i(q)$ is given by the sum of the two direct blocking factors $DB_{i,lpr}$ in (3.21) and $DB_{i,hpr}$ in (3.22).

$$BT_i(w_i(q)) = DB_{i,lpr}(w_i(q)) + DB_{i,hpr}(w_i(q)) \tag{3.23}$$

For each task τ_i, the blocking time equation (3.23) is part of the busy-window iterative computation that have to be solved in order to bound the task's worst-case response time under partitioned multi-core SPNP scheduling.

3.8.2 Response Time Analysis for Partitioned Multi-Core SPNP Scheduling

Relying on the system model introduced in Section 3.3 and on the background provided by the analysis approach for single-core non-preemptive systems discussed in Section 3.5.1, in this section we introduce the response time analysis approach for arbitrarily activated tasks scheduled non-preemptively in partitioned multi-core systems with shared resources.

For this, two important aspects need to be considered. Firstly, in single-core non-preemptive setups the influence of sharing resources can be neglected due to the intrinsic behavior of the non-preemptive scheduler which avoids the synchronization overhead due to resource sharing mechanisms (see Section 3.5.1). In multi-core systems this is not the case anymore. The blocking times that tasks will experience due to conflicting accesses (see Figure 3.12 in Section 3.8.1) for shared resources have to be computed and considered when deriving response times bounds.

Secondly, the critical instant scenario (see Figure 3.8b) in Section 3.5.1), on which the classical response-time analysis procedure rely, must be revisited. It is known that the use of global shared resources may lead to suspension of tasks which possibly defers the task execution times and thus counters the assumptions regarding the critical instant scenario on which the classical response time analysis approach rely (see [116]). This aspect, which was identified in case of sharing resources under multiprocessor static-priority preemptive scheduling, will be now investigated for the case of sharing resources under multiprocessor static-priority non-preemptive scheduling.

3.8.2.1 Critical Instant

If global resources are not shared between tasks mapped on different cores, the response time analysis problem reduces to the classical approach discussed in Section 3.5.1. There, a task τ_i experiences the critical instant scenario, which leads to the worst-case response time, when it is released (i) at the time moment just after a job of a lower priority local

task $\tau_j \in lpl(i)$ has started its local execution and (ii) simultaneously with jobs of all higher priority local tasks (tasks $\in hpl(i)$).

These arguments also hold when resources are shared between tasks mapped on different cores. Relying on the resource arbitration policy introduced in Section 3.8.1 a task which has an outstanding request for a shared resource will actively wait for that resource without any preemption by other local tasks. This means that the blocking times due to the waiting for shared resources represent an extension of the task's core execution time. Thus, in case of inter-core synchronization mechanisms and core local non-preemptive scheduling, for any job J_i there can not be any higher priority local job that can suspend itself when waiting for a global shared resource. Therefore the effect of deferred execution [116], identified in case of suspension based shared resource arbitration, does not counter the assumptions regarding the critical instant scenario in case of spinning based resource arbitration and non-preemptive core scheduling.

A job J_j of a task τ_j with lower priority than τ_i will start its execution on its host core and will possibly request and even lock required shared resources only when there is no other previously released and unfinished job of a task with priority higher than τ_j. Thus, a task τ_j can delay the execution of a higher priority local task τ_i only if it starts executing before the release time of task τ_i and before the release of any other local task with priority higher than i.

From the perspective of the higher priority local tasks, similar to the single-core analysis approach, these will cause the largest possible delay for a local task τ_i if they are released simultaneously with task τ_i.

The critical instant for a task τ_i under partitioned SPNP multi-core scheduling is represented in Figure 3.13 where task τ_i is activated at the same moment with the higher priority tasks τ_{hp1} and τ_{hp2} just after the lower priority task τ_{lp} has started its execution. The terms BT in Figure 3.13 represent the blocking times that different jobs running on a core may experience when the requested global shared resources are locked by jobs of the remote tasks.

3.8.2.2 Derivation of the Maximum Level-i Busy Window

Similar to the analysis for uni-processor static priority non-preemptive scheduling the worst-case response time of a task τ_i non-preemptively scheduled in multi-core systems with shared resources is given by the largest response time of any of the q ($q = 1 \dots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the maximum level-i busy window. Assuming the critical instant scenario under non-preemptive scheduling in a multi-core setup, the maximum level-i busy window $w_i(q)$ of a task τ_i consists not only of the time intervals during which the tasks contributing to the busy window execute but also of the time intervals these tasks are blocked and have to wait for the required global shared resources (see Figure 3.13). According to the blocking time analysis introduced in Section 3.8.1 the blocking time of a task in multi-core systems with global shared resources is a function of the window size $w_i(q)$ during which the task initiates requests to the required shared resources. Thus, the length of the level-i busy window of a task τ_i in a

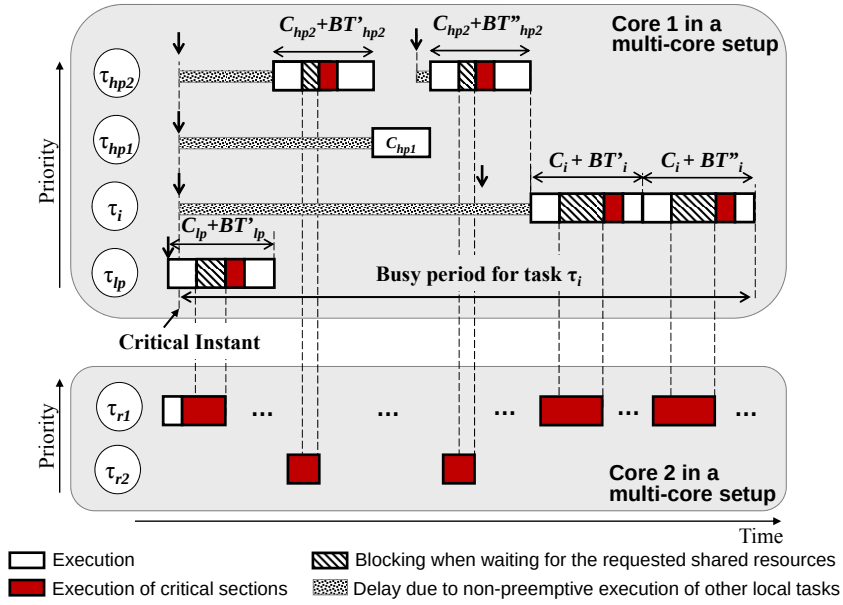


Figure 3.13: Critical instant and busy window for a task τ_i in a partitioned multi-core system with cores individually scheduled according to the SPNP scheduling.

multi-core system is composed of:

1. the longest possible initial blocking (denoted with LB_i) caused by one instance of a lower priority local task due to the non-preemptive scheduling behavior. In case of multi-core setups under SPNP scheduling, the initial blocking time caused by one of the lower priority local tasks is composed of the task's core execution time plus the blocking time when waiting for global shared resources. This is given by:

$$LB_i(w_i(q)) = \max_{\forall \tau_j \in lpl(i)} (C_j + BT_j(w_i(q))) \tag{3.24}$$

   where $BT_j(w_i(q))$ is the blocking time of a job of task τ_j in a time window $w_i(q)$ (see (3.23) in Section 3.8.1). This is given by :

2. the execution of jobs of task τ_i and of the tasks with priority higher than the priority of task τ_i, i.e. jobs J_j of tasks $\tau_j \in hep(i)$ where $hep(i) = \tau_i \bigcup hpl(i)$ [11], plus the blocking time these jobs will suffer when accessing global shared resources. This is given by :

$$\sum_{\forall \tau_j \in hep(i)} (\eta_j^+(w_i(q)) \cdot C_j + BT_j(w_i(q))) \tag{3.25}$$

The maximum level-i busy window L_i of a task τ_i in partitioned multi-core systems under SPNP scheduling can be calculated with the following recurrence relation:

[11] Tasks have unique priorities (see Section 3.3).

$$L_i^{n+1} = LB_i(L_i^n) + \sum_{\forall \tau_j \in hep(i)} (\eta_j^+(L_i^n) \cdot C_j + BT_j(L_i^n)) \tag{3.26}$$

In comparison to equation (3.1) for single-core processors, equation (3.26) contains new components, i.e. the blocking time derivation, which challenge the classical iterative calculation procedure. As presented in [130] and [88] and as will be detailed in Section 3.10 all components of equation (3.26) grow monotonically with respect to the window size and therefore allow the iterative calculation of a solution for (3.26).

The recurrence relation (3.26) starts with an initial value $L_i^0 = C_i$, and finishes when $L_i^{n+1} = L_i^n$ (i.e. two consecutive iterations provide identical results). The recurrence relation in the uni-processor analysis was guaranteed to converge if the resource utilization was less than 100%. In comparison to single-core systems, in multi-core setups under SPNP scheduling the utilization of each individual core is a function not only of the tasks' core execution times but also of the blocking times. Thus, the iterative calculation has to be stopped if the "effective" core utilization level (composed of the core execution times and blocking times of the tasks) exceeds 100% at some iteration point. In that case the task set is considered unschedulable.

3.8.2.3 Derivation of the Worst-Case Response Times

Similar to the analysis approach for single-core processors, the number of task instances that have to be considered when computing the worst-case response time of a task τ_i under partitioned multiprocessor SPNP scheduling is given by $Q_i = \eta_i^+(L_i)$, with L_i obtained with (3.26) above.

Thus, the worst-case response time of a task τ_i is given by the largest response time of any of the q ($q = 1 \ldots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the busy window $w_i(q)$ as follows:

$$\begin{aligned} R_i &= \max_{q=1\ldots Q_i} r_i(q) \\ r_i(q) &= w_i(q) + C_i - \delta_i^-(q) \end{aligned} \tag{3.27}$$

where the maximum level-i busy window $w_i(q)$ of the $(q-1)$-th activation (i.e the queueing delay of the q-th activation) is generally computed with (3.7), which is:

$$w_i^{n+1}(q) = (q-1) \cdot C_i + BT_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q)) \cdot C_j$$

Because in multi-core systems the blocking time term BT_i is a function of the busy-window and comprises multiple blocking factors, the equation above can be rewritten for partitioned multi-core systems under SPNP scheduling and MLP-NP shared resource arbitration as

$$\begin{aligned} w_i^{n+1}(q) &= (q-1) \cdot C_i + LB_i(w_i^n(q)) + BT_i(w_i^n(q)) \\ &\quad + \sum_{\forall \tau_j \in hpl(i)} (\eta_j^+(w_i^n(q)) \cdot C_j + BT_j(w_i^n(q))) \end{aligned} \tag{3.28}$$

where $LB_i(w_i^n(q))$ is the initial local blocking time (computed with (3.24)) caused by the lower priority tasks that may be executed when τ_i becomes ready for execution; $BT_i(w_i^n(q))$ is the direct blocking time (given by (3.23)) of task τ_i when waiting for the required global shared resources; $hpl(i)$ is the set of tasks with priority higher than i; $\eta_j^+(w_i^n(q))$ is the maximum amount of jobs of task τ_j in a time window of size $w_i^n(q)$; and $BT_j(w_i^n(q))$ is the direct blocking time (also given by (3.23)) that jobs of task τ_j will experience in the analysed time window $w_i^n(q)$.

Similar to equation (3.26), the recurrence relation (3.28) can be solved by iteration, because all components grow with the window size [130, 88]. The recurrence starts with a value of $w_i^0(q) = q \cdot C_i$ and ends when $w_i^{n+1}(q) = w_i^n(q)$, or when the value $w_i^n(q)$ at some iteration point is so large that the obtained response time r_i for the current considered activation q already exceeds τ_i's deadline, in which case the task is unschedulable.

Finally, if worst-case response time values R_i have been obtained for all the tasks in the multi-core system, the schedulability test consists of checking whether the condition $R_i \leq D_i$ holds for every task τ_i.

3.9 Response-Time Analysis for AUTOSAR conform Multi-Core ECUs

The previous two sections independently addressed the timing analysis of partitioned multi-core setups with shared resources under SPP (Section 3.7) and SPNP (Section 3.8) scheduling. Nevertheless, the combination of both is of particular relevance for the next generation of AUTOSAR conform automotive multi-core ECUs where preemptive and non-preemptive scheduling will co-exist on each core. This section addresses this subject and presents a novel analysis method which allows the calculation of response-times for tasks with arbitrary activations and deadlines which share resources in multi-core systems scheduled according to the partitioned fixed-priority AUTOSAR OS [12]. With this we cover the current and foreseeable automotive practice regarding standards (OSEK, AUTOSAR), priority assignments (static and often manually assigned), and inter-task synchronization (through a lock-based mechanism).

In order to introduce the AUTOSAR OS aware timing analysis solution, in Section 3.9.1 we first extend the multi-core system model from Section 3.3 with automotive specific elements and introduce the complete scheduling model of automotive applications. After that, in Section 3.9.2 we address the the AUTOSAR spinlock-based resource arbitration mechanism in the context of multi-core AUTOSAR OS scheduling. Further, based on the shared resource load derivation in Section 3.6 we introduce the corresponding blocking-time analysis. Finally, in Section 3.9.3, the blocking time equations will be integrated in the response-time analysis procedure for AUTOSAR conform multi-core systems scheduled according to the partitioned fixed-priority AUTOSAR OS.

3.9.1 Extended Multi-Core System and Scheduling Model

According to the system model introduced in Section 3.3 we consider a set of real-time applications statically mapped on a set of m ($m \geq 2$) processor cores. Each application

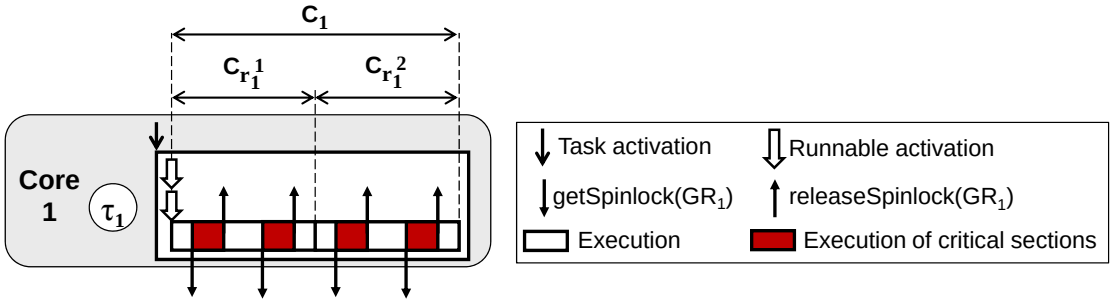


Figure 3.14: Example of a task instance with two equally long runnables, each performing two requests for GRs.

is composed of one or multiple arbitrarily activated tasks, each instance (job) of a task being considered activated by an internal or external system event.

In automotive applications, tasks are usually composed of multiple so called runnables (can be seen as subtasks). This means that each job J_i of a task τ_i may be composed of multiple runnables r_i^k, $k = 1 \dots K_i, K_i \in \mathbb{N}^+$, with K_i being the maximum number of runnables of each job J_i of a task τ_i. Each runnable r_i^k is characterized by its worst-case execution time C_{r_i}. For convenience we assume all runnables of a task instance to be of equal size [12]. Thus, the worst-case execution time of each job J_i of a task τ_i is $C_i = K_i * C_{r_i}$. Runnables inherit the tasks activation pattern and are executed in order [13], i.e. $r_i^1, r_i^2, \dots r_i^k$. In other words, each time a job J_i is activated one has a burst of K_i activations of the job's runnables.

During execution, each job can perform multiple accesses to local (LRs) and global resources (GRs). Shared resources, which are visible to users and addressable through system calls [14], are assumed to be objects that require serialized access. Each of these accesses is considered a critical section guarded by a semaphore and protecting a LR or a GR. We differentiate between local critical sections (lcs) or global critical sections (gcs).

As jobs are composed of one or multiple runnables, each of these runnables is assumed to perform accesses to LRs and GRs. In this case we model the number and the size of critical sections per runnable similar to an usual job J_i. Thus, for each runnable r_i^k the maximum number of gcs is $n_{r_i}^G$. As all runnables of a task are assumed identical, for any job J_i, comprising k runnables r_i^k, we have the maximum number of gcs $n_i^G = k * n_{r_i}^G$. An example of an instance of task τ_1 composed of two equally long runnables, each performing two requests for GRs is depicted in Figure 3.14.

[12] This assumption does not constrain the analysis capabilities. If runnables of different sizes would be modelled, the analysis equations should always consider the delays and the blocking caused by the largest runnable of each task. This would lead to pessimistic but conservative results.

[13] In practice, some of the runnables might not be activated when the task is activated, the order of those activated being preserved. However, in this thesis we are exclusively interested in the worst-possible scenario, i.e. when all runnables of each task are always activated.

[14] The API calls getResource/releaseResource are used for addressing LRs [100]. GRs are addressed through the API calls getSpinlock/releaseSpinlock [12].

Scheduling Model.

On each core of the multi-core ECU resides an independent OSEK/AUTOSAR scheduler according to which tasks are locally scheduled. The OSEK OS [100] and therewith the AUTOSAR OS [12] allows three types of scheduling: *fully preemptive*, *fully non-preemptive* and *mixed-preemptive*. The *OSEK mixed-preemptive* scheduling supports a mixture of *preemptive*, *non-preemptive* and *cooperative* scheduling and is de facto implemented in the current automotive ECUs [100].

More exactly:

- The default scheduling procedure on an ECU is preemptive scheduling.
- Beside this, the operating system allows tasks to combine aspects of preemptive and non-preemptive scheduling by defining groups of tasks. In order to schedule tasks non-preemptively the automotive standards OSEK and AUTOSAR allow tasks to be arranged in groups. This means that several tasks on each core can be grouped together such that they share a group internal resource (one virtual/logical, not necessarily physical resource per group). Tasks within a group behave as non-preemptive to each other. Group internal resources are arbitrated according to the PCP [100], are not accessible to the user and can therefore not be addressed but are strictly managed internally. Multiple groups can be defined per core where each group is composed of tasks with adjacent priorities, i.e. there is no task that has lower priority than a task of a group and higher priority than another task of a group without being itself part of that group.
- Additionally, tasks within a group can be scheduled cooperatively, i.e. they are by default non-preemptive to each other but, by explicitly calling the RESCHEDULE interface [100] at specific scheduling points, usually at runnables borders [15], the running task releases the group's internal resource. Therewith it allows the highest priority task in the group, which is ready for execution, to lock the internal resource and further execute non-preemptively. Note that, all tasks in a group are configured either as non-preemptable or as cooperative. In other words, inside a group of tasks there cannot be a mixture of non-preemptive and cooperative scheduling.

Tasks that are not part of any group can always preempt lower priority tasks, even those that are part of a group. Similarly, tasks in different independent groups can always preempt each other based on their priorities.

For the arbitration of shared resource accesses we consider: for LRs the Priority Ceiling Protocol (PCP) specified in the OSEK standard [100] [16] and for GRs the AUTOSAR spinlock-based mechanism [12].

[15] In the automotive applications the rescheduling points are usually at the runnables borders. Further details on this are beyond the scope of this thesis.

[16] The implementation version of the Priority Ceiling Protocol [116] specified in the OSEK standard is known in literature as the Immediate Priority Ceiling Protocol (IPCP).

Example of Multi-Core ECU.

An example dual-core ECU system is depicted in Figure 3.15. Each core is running five tasks that are numbered in the order of their priority. The tasks are locally scheduled according to AUTOSAR scheduling policy as follows:

- on Core 1 task τ_1 has the higher priority and may preempt any of the lower priority local task; the tasks in the two groups - the first group comprising the tasks τ_4 and τ_6 and the second group comprising τ_7 and τ_9 - are scheduled non-preemptively or cooperatively. However, the tasks τ_4 and τ_6 may preempt the execution of the tasks τ_7 and τ_9;
- on Core 2 task τ_2 has the higher priority and may preempt any of the lower priority local tasks; τ_3, τ_5 and τ_8 are arranged in a group and thus are non-preemptive or cooperative to each other; task τ_{10} has the lowest priority and can be preempted by any of the higher priority local tasks.

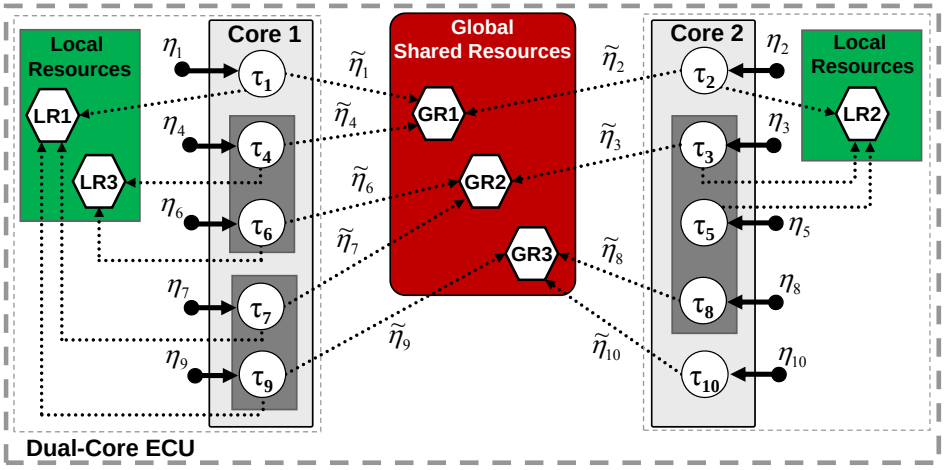


Figure 3.15: Dual-core ECU with tasks accessing local and global shared resources.

The local shared resources are $LR1$ and $LR3$ for Core 1 and $LR2$ for Core 2. The shared resources which are used by tasks mapped to different cores are the three global shared resources $GR1$, $GR2$ and $GR3$.

The task activating event models are denoted with η_1 to η_{10}, where the index identifies the activated task. The corresponding loads imposed on the shared resources are denoted with $\tilde{\eta}$, e.g. with $\tilde{\eta}_1$.

The difference between fully non-preemptive and cooperative scheduling inside a task group is illustrated in Figure 3.16a) where τ_4 is blocked by the size of the τ_6's core execution time (i.e. by all runnables) and Figure 3.16b) where τ_4 preempts the execution of τ_6 after the completion of its first runnable. Figure 3.16 illustrates an scheduling example for tasks τ_1, τ_4 and τ_6 on Core 1 under the assumption they are not requesting any shared resource and the other tasks on the core are not activated.

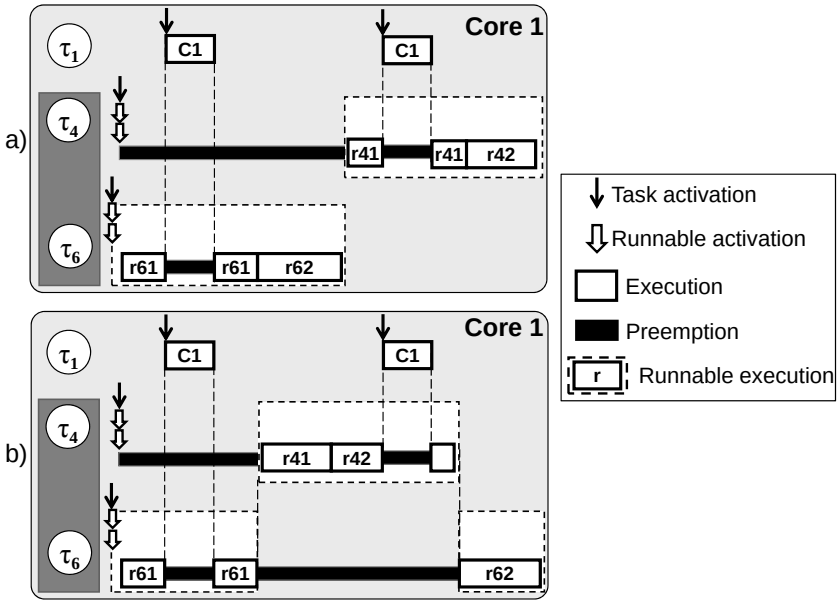


Figure 3.16: Scheduling example on Core 1 where tasks τ_4 and τ_6 a) are fully non-preemptive and b) are cooperative to each other.

3.9.2 Blocking Time Analysis for AUTOSAR conform Multi-Core ECUs

In the following we consider the AUTOSAR specific arbitration policies for shared resources (i.e. the PCP and the spinlock-based mechanisms) in the context of partitioned AUTOSAR conform multi-core scheduling (i.e. preemptive, non-preemptive and cooperative) and introduce the corresponding blocking time analysis for task sets with arbitrary activation patterns (event models).

3.9.2.1 Specification of the AUTOSAR Shared Resource Arbitration Policy

- **Arbitration of local shared resources.** For the arbitration of local resources the AUTOSAR OS uses on individual cores the priority ceiling protocol PCP inherited from the single-core OSEK OS [100]. According to PCP specified in the OSEK standard each semaphore associated to a LR is allocated offline a static priority ceiling which is equal to the highest priority of all tasks which access that LR. At runtime, when a a job J_i of a task τ_i locks the semaphore corresponding to LR it immediately inherits its associated priority ceiling. Thus, the *lcs* corresponding to the locked LR is executed at the level of the offline assigned priority ceiling. This implementation version of the Priority Ceiling Protocol is known as Immediate Priority Ceiling Protocol (IPCP).

- **Arbitration of global shared resources.** For the arbitration of GRs the AUTOSAR OS uses a spinlock-based arbitration mechanism [12] as follows: during execution, a task τ_i may request a certain GR [17] and will actively wait (spin) if this is

[17] by using one of the APIs TryToGetSpinlock() or GetSpinlock() [12].

occupied by a remote task; during active waiting a task may be preempted by higher priority local tasks, but lower priority local tasks cannot start executing; if a task locks a GR it suspends all interrupts [18] on his host core and thus it becomes non-preemptable. As AUTOSAR does not provide implementation details of the APIs for addressing resources and disabling interrupts we assume that interrupts will be atomically disabled as part of the software construct for the lock acquisition.

Following the principle Cooperate on standards, compete on implementation the AUTOSAR specifications does not contain further implementation details regarding the multi-core synchronization protocol. In particular, AUTOSAR does not specify any semantic for the case of coinciding requests initiated by multiple jobs running on diffent cores for a certain GR. However, from Section 3.4 we know that the order of granting the locks is one essential design decision that must be specified in order to ensure predictable upper bounds on the tasks timing behavior. Therefore, for the purpose of this thesis we assume the following implementation considerations: associated with each global semaphore is a priority-ordered queue of tasks that busy-wait on the semaphore. If a task needs to lock a global resource and this is currently held by another task, the task queues itself on the semaphore queue. Thus, in case of multiple coinciding requests for a certain global shared resource, the highest priority job requesting it will get the lock on the associated semaphore [19]. If a task is preempted while busy-waiting its request is cancelled and will be removed from the semaphore queue. This task will queue up again for the required resource when scheduled again on the host processor core [20].

- **Nested calls for shared resources.** In order to avoid deadlocks, nested accesses to shared resources are not allowed [21].

3.9.2.2 Derivation of Blocking Time

The blocking time derivation for the AUTOSAR spinlock-based synchronisation mechanism follows the blocking time derivation for the MPCP protocol under SPP multiprocessor scheduling in Section 3.7.1 and for the MLP-NP protocol under SPNP multiprocessor scheduling in Section 3.8.1. Thus, the blocking time equations we introduce next capture the blocking scenarios for each task τ_i that accesses local and global resources and is activated q times in a time window of size $w_i(q)$, i.e. in the maximum level-i busy window. The calculation of the maximum level-i busy window for the AUTOSAR conform multiprocessor scheduling will be discussed in Section 3.9.3.

[18] with the API SuspendAllInterrupts() [12].

[19] As priority based execution is state-of-the art in the automotive design, for the purpose of this thesis we assume that locks are assigned based on tasks priorities and thus maintain the compatibility with the priority based scheduling on the individual cores. An evaluation of other design options of the arbitration protocol, e.g. FIFO based resource locking, is beyond the scope of this thesis.

[20] If the requests of the preempted tasks would remain in the priority-queue associated to the semaphores, the interference of the preempting tasks shall be reflected in the blocking time of other remote tasks trying to access the same shared resource. In that case, the blocking time would be a function of the normal task execution and not a function of their critical sections.

[21] If nesting is required, an explicit partial ordering of calls for GRs has to be predefined offline in order to avoid deadlocks and potential starvation situations.

In order to introduce the blocking factors, we refine the list of the parameters and the terminology in Table 3.1. We previously defined $lpl(i)$ and $hpl(i)$ as the sets of tasks mapped on the same core as τ_i which have lower and higher priority than τ_i. In case of AUTOSAR scheduling setups we have:

- the set $lpl(i)$ which comprises the set of the lower priority local preemptable tasks $lplP(i)$, the set of the lower priority non-preemptable tasks $lplNP(i)$ and the set of the lower priority local cooperative tasks $lplC(i)$.
- the set $hpl(i)$ which comprises the sets of the higher priority local tasks to which τ_i is preemptable $hplP(i)$, non-preemptable $hplNP(i)$ and cooperative $hplC(i)$.
- the set $\Psi(i)$ which contains higher priority local tasks to τ_i, except those to which τ_i is non-preemptable, i.e. $\Psi(i) = hpl(i) \setminus hplNP(i)$.

Under AUTOSAR scheduling and AUTOSAR shared resource arbitration four blocking scenarios have to be considered for each analyzed task:

1. **Direct remote blocking** - each task τ_i can be blocked when trying to access a GR if this has already been locked by a remote task with lower or higher priority.
2. **Indirect blocking** - in case a task τ_i is preempted by higher priority local tasks and these are blocked by remote tasks, the blocking time of the higher priority tasks prolongs the blocking of task τ_i.
3. **Blocking when re-initiating cancelled requests for global resources** - in case a task τ_i or the higher priority local tasks to τ_i are preempted by other higher priority local tasks while busy-waiting, these tasks will re-initiate the cancelled requests for the required shared resource after being rescheduled on the core. Each of the re-initiated requests can be blocked by a request of a remote task.
4. **Local blocking** - each task τ_i can be blocked by a lower priority local task if this can temporarily be non-preemptive. Therefore, under AUTOSAR OS it is key to differentiate between the different types of lower priority local tasks, i.e. preemptive, non-preemptive and cooperative.

Analysis equations for deriving the blocking times that a job J_i of a task τ_i can experience in the above enumerated blocking scenarios will be introduced next.

1. Direct Blocking Time. According to the AUTOSAR specification a task that tries to access a GR can either lock it, if the resource is available, or it will start "busy-waiting" (i.e. spinning) until the resource becomes available. As the GR can be accessed by multiple tasks running on the remote processor cores it is important to explicitly distinguish between dual-core (i.e. $m = 2$) and multi-core (i.e. $m > 2$) architectures. Depending on the multi-core applications and on their mapping, the differentiation between dual-core and multi-core setups helps ruling out some blocking scenarios which leads to reduced blocking times.

1.1. Direct Blocking Time in Multi-Core Systems. When a job J_i of a task τ_i requests a GR this can be locked by a lower priority job J_j of a task mapped on a different core than τ_i. In the worst-case scenario, each time when J_i attempts to lock

a GR, it may find that this is currently locked by a lower priority job on another core (i.e. by one of the jobs J_j of the tasks in $\theta_{i,j}$). Thus, each request for a global shared resource of a job J_i can be blocked for the duration of the longest global critical sections ω_j^{GR} of a lower priority remote job.

The blocking time due to *lpr* tasks which share the same global resources with J_i can be generally calculated with:

$$DB_{i,lpr}^{MC}(w_i(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j}} (\omega_j^{GR}) \tag{3.29}$$

Similar to the previous blocking factor, each job J_i can be blocked by higher priority remote jobs that request the same global resource as J_i (i.e. jobs of tasks in the set $\Theta_{i,j}$). As opposed to lower priority remote jobs, higher priority remote jobs may be served multiple times before jobs of task τ_i will be able to lock the requested GRs.

$$DB_{i,hpr}^{MC}(w_i(q)) = \sum_{\forall \tau_j \in \Theta_{i,j}} (\tilde{\eta}_j^+(w_i(q)) \cdot \omega_j^{GR}) \tag{3.30}$$

The worst-case direct blocking time $DB_i^{MC}(w_i(q))$ a task τ_i can encounter in a time window $w_i(q)$, when executing on a multi-core system with m ($m > 2$) cores, is given by the sum of the two blocking factors in (3.29) and (3.30):

$$DB_i^{MC}(w_i(q)) = DB_{i,lpr}^{MC}(w_i(q)) + DB_{i,hpr}^{MC}(w_i(q)) \tag{3.31}$$

1.2. Direct Blocking Time in Dual-Core Systems. While in multi-core setups with more than two cores several tasks can compete for the same GR, in a dual-core system only two tasks mapped on different cores can simultaneously compete for a GR. Thus, in the worst-case each request of a job J_i of a task τ_i for a GR will be blocked by a remote task with higher or lower priority than J_i. In comparison to multi-core setups, in a dual-core system the waiting job J_i will lock the required GR as soon as this is released by the remote task. The worst-case blocking time $DB_i^{DC}(w_i(q))$ a task τ_i can encounter in a time window $w_i(q)$ when executing on a dual-core system (i.e. $m = 2$ cores) can be calculated with:

$$DB_i^{DC}(w_i(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j} \bigcup \Theta_{i,j}} \left(\omega_j^{GR}\right) \tag{3.32}$$

2. Indirect Blocking Time. In case a task τ_i is preempted by a *hpl* task and this gets blocked, the blocking time of the *hpl* task prolongs the delay of task τ_i. According to the AUTOSAR specification a task keeps spinning for the requested GR until the resource becomes available or a *hpl* task preempts it. Thus, a preempted task τ_i cannot execute for the time *hpl* tasks are blocked [22]. However, not all of the *hpl* tasks can

[22] Of course a preempted task cannot execute not only for the time higher priority local tasks are busy-waiting but also for the time these tasks are normally executing. However, the normal execution of higher priority tasks is captured in the response-time analysis as higher priority interference and not as blocking time (see Section 3.9.3).

preempt task τ_i. If task τ_i is part of a group, only tasks not being within that group and those belonging to that group but specified as cooperative can preempt τ_i, i.e. tasks in $\Psi(i)$. All these aspects have to be considered when deriving the terms for the indirect blocking. Furthermore, similar to the direct blocking time, the indirect blocking time depends on the number of cores in the system.

2.1. Indirect Blocking Time in Multi-Core Systems. As already known from the direct blocking scenarios considered above (see **1.1.**), in case of multi-core architectures with $m > 2$ a task can be blocked several times by multiple remote tasks. This holds not only for the analyzed task τ_i but also for the higher priority local tasks which can preempt τ_i, i.e. $\tau_k \in \Psi(i)$, during its execution outside critical sections or during busy-waiting. Similar to τ_i, requests for global resources of each job J_k of *hpl* tasks $\tau_k \in \Psi(i)$ can be directly blocked by remote tasks with lower or higher priority, i.e. by tasks $\tau_j \in \theta_{k,j} \bigcup \Theta_{k,j}$. Thus, the indirect blocking time a task τ_i will experience in a multi-core setup due to the direct blocking of the *hpl* tasks $\tau_k \in \Psi(i)$ can be derived with an equation similar to (3.31) as follows:

$$\begin{aligned} IB_i^{MC}(w_i(q)) &= \sum_{\forall \tau_k \in \Psi(i)} DB_k^{MC}(w_i(q)) \\ &= \sum_{\forall \tau_k \in \Psi(i)} (DB_{k,lpr}^{MC}(w_i(q)) + DB_{k,hpr}^{MC}(w_i(q))) \\ &= \sum_{\forall \tau_k \in \Psi(i)} (\eta_k^+(w_i(q)) \cdot n_k^G \cdot \max_{\forall \tau_j \in \theta_{k,j}} (\omega_j^{GR}) + \sum_{\forall \tau_j \in \Theta_{k,j}} (\tilde{\eta}_j^+(w_i(q)) \cdot \omega_j^{GR})) \end{aligned} \tag{3.33}$$

In other words, the indirect blocking time of a task τ_i is given by the direct blocking times of the higher priority local tasks that can preempt the analyzed task τ_i.

2.2. Indirect Blocking Time in Dual-Core Systems. In comparison to setups with more than 2 processor cores, in dual-core setups, each request for a global resource of each job J_k of *hpl* tasks that may preempt task τ_i (i.e. $\tau_k \in \Psi(i)$) can be blocked by only one remote request of a task with either lower or higher priority, i.e. by only one job of a task $\tau_j \in \theta_{k,j} \bigcup \Theta_{k,j}$. As each job J_k performs n_k^G requests for global resources and in a time window $w_i(q)$ there can be at most $\eta_k^+(w_i(q))$ jobs of task τ_k, the indirect blocking time a task τ_i will experience in a dual-core setup can be calculated with:

$$IB_i^{DC}(w_i(q)) = \sum_{\forall \tau_k \in \Psi(i)} \eta_k^+(w_i(q)) \cdot n_k^G \cdot \max_{\forall \tau_j \in \theta_{k,j} \bigcup \Theta_{k,j}} \left(\omega_j^{GR}\right) \tag{3.34}$$

3. Blocking when re-initiating cancelled Requests for Global Resources. Each time a job J_i of the analyzed task τ_i is preempted while busy-waiting, its request for the global resource is cancelled. At the moment when J_i is re-scheduled and re-initiates the request for the global resource, it may be blocked by a remote job that

could acquire the lock while J_i was preempted. Two aspects have to be considered in order to find an upper bound for this blocking type, namely (i) the maximum number of requests a task τ_i can re-initiate and (ii) the maximum time each of the re-initiated requests can be blocked:

(i) Regarding the maximum number of re-initiated requests of a task τ_i this is given by the maximum number of preemptions this task can experience during its busy window $w_i(q)$. Because in the context of AUTOSAR scheduling τ_i may not be preemptable to all higher priority local tasks, one has to consider only those higher priority tasks that can preempt τ_i, i.e. $\tau_k \in \Psi(i)$. However, cooperatively scheduled tasks, if any configured in the system, permit preemptions only at their runnables borders but not during busy-waiting. Therefore, preemptions by the higher priority local tasks to which τ_i is cooperative (i.e. tasks in $hplC(i)$) don't have to be considered in this blocking factor. Thus, the maximum number of preemptions during busy waiting of jobs of task τ_i in a time window $w_i(q)$ is given only by tasks in $hplP(i)$ as follows:

$$\sum_{\forall \tau_k \in hplP(i)} \eta_k^+(w_i(q))$$

(ii) Regarding the maximum time each of the re-initiated requests can be blocked one has to identify the tasks that cause this blocking. In general, requests for global shared resources can be blocked by lower and higher priority remote tasks. As known from the direct blocking scenario, the influence of the remote tasks depends of the number of cores in the system.

3.1. Blocking Time due to re-initiated Requests in Multi-Core Systems. In multi-core setups each request for a global shared resource can be blocked once by one global critical section of a lower priority remote task and multiple times by global critical sections of higher priority remote tasks.

In a worst-case scheduling scenario, each re-initiated request of task τ_i or of the tasks that can preempt τ_i (i.e. $\tau_k \in hplP(i)$) can be blocked once by a lower priority remote task in the sets $\theta_{i,j}$ or $\theta_{k,j}$ [23] for the duration of the longest global critical section $\max_{\forall \tau_j \in \theta_{i,j} \bigcup \theta_{k,j}} \left(\omega_j^{GR}\right)$.

The influence of the higher priority remote tasks on task τ_i and on its higher priority local tasks $\tau_k \in hplP(i)$ is safely upper bounded in the direct blocking time - see (3.30) - and in the indirect blocking time (i.e. in the direct blocking time of the higher priority local tasks) - see right hand side term in (3.33) - independent on the number of re-initiated requests.

[23] In order to reduce a potential overestimation, the highest priority task that can preempt τ_i has to be excluded from the set $\theta_{k,j}$. This is because the highest priority task in $hplP(i)$ can preempt the execution of τ_i but its requests won't be re-initiated and thus not additionally blocked by a lower priority remote task.

Thus, the maximum possible blocking time of a task τ_i that results from τ_i or its higher priority local tasks being preempted while busy-waiting is captured by:

$$CRB_i^{MC}(w_i(q)) = \sum_{\forall \tau_k \in hplP(i)} \eta_k^+(w_i(q)) \cdot \max_{\forall \tau_j \in \theta_{i,j} \bigcup \theta_{k,j}} \left(\omega_j^{GR}\right) \tag{3.35}$$

3.2. Blocking Time due to re-initiated Requests in Dual-Core Systems. In comparison to setups with more than 2 processor cores, in dual-core setups, each request for a global resource of each job of task τ_i and of the higher priority tasks $\tau_k \in hplP(i)$ can be blocked by only one remote request of a task with either lower or higher priority, i.e. by only one critical section of a task $\tau_j \in \theta_{i,j} \bigcup \Theta_{i,j} \bigcup \theta_{k,j} \bigcup \Theta_{k,j}$. Thus, the maximum possible blocking time of a task τ_i that results from τ_i or its higher priority local tasks being preempted while busy-waiting is captured by:

$$CRB_i^{DC}(w_i(q)) = \sum_{\forall \tau_k \in hplP(i)} \eta_k^+(w_i(q)) \cdot \max_{\forall \tau_j \in \theta_{i,j} \bigcup \Theta_{i,j} \bigcup \theta_{k,j} \bigcup \Theta_{k,j}} \left(\omega_j^{GR}\right) \tag{3.36}$$

4. Local Blocking Time. According to the uniprocessor priority ceiling protocol (PCP) a job J_i of a task τ_i can be blocked once by a *lpl* job J_k (i.e. $\tau_k \in lpl(i)$) if this can be temporarily non-preemptive. Under AUTOSAR OS it is essential to differentiate between the different types of *lpl* tasks, i.e. preemptive, non-preemptive and cooperative.

4.1. Local Blocking Time due to Preemptive Tasks. A *lpl* task τ_k is preemptable for a task τ_i if τ_k is *not* part of the same group as τ_i. Such a *lpl* task can block task τ_i for the duration of a *lcs* or *gcs* it locks, i.e. ω_k^{LR} or ω_k^{GR}. Thus, the local blocking time $LB_i^P(w_i(q))$ of a task τ_i due to preemptive *lpl* tasks, i.e. $\tau_k \in lplP(i)$ is given by the maximum length of a local or of a global critical section as follows:

$$LB_i^P(w_i(q)) = \max_{\forall \tau_k \in lplP(i)} \left\{\omega_k^{\mathrm{LR}}, \omega_k^{\mathrm{GR}}\right\} \tag{3.37}$$

In case of overlapping activations of task τ_i, a lower priority local preemptable job J_k will block only the first job of task τ_i. As tasks under AUTOSAR OS do not suspend when waiting for GRs, once J_k exits the critical section which blocks τ_i it won't execute anymore before all activated jobs of τ_i are finished.

4.2. Local Blocking Time due to Non-Preemptive Tasks. Lower priority local tasks within the same group as τ_i, configured as non-preemptive (i.e. tasks in $lplNP(i)$), can not be preempted by τ_i at any point. Consequently a task $\tau_k \in lplNP(i)$ blocks τ_i only once with its whole WCET C_k plus the time it is directly blocked by other tasks on other cores during the time interval C_k.

$$LB_i^{NP}(w_i(q)) = \max_{\forall \tau_k \in lplNP(i)} \{C_k + DB_k(C_k)\} \tag{3.38}$$

where $DB_k(C_k)$ depends on the number of cores in the system according to (3.31) or (3.32), i.e.:

$$DB_k(C_k) = \begin{cases} DB^{MC}_{k,lpr}(C_k) + DB^{MC}_{k,hpr}(C_k); \text{ if } m > 2 \\ DB^{DC}_{k}(C_k); \text{ if } m = 2 \end{cases}$$

As only one job of a lower priority local non-preemptable task can delay the execution of the analyzed task τ_i the equations above can be rewritten as

$$DB_k(C_k) = \begin{cases} 1 \cdot n^G_k \cdot \max\limits_{\forall \tau_j \in \theta_{k,j}} (\omega^{GR}_j) + \sum\limits_{\forall \tau_j \in \Theta_{k,j}} (\tilde{\eta}^+_j(C_k) \cdot \omega^{GR}_j)); \text{ if } m > 2 \\ 1 \cdot n^G_k \cdot \max\limits_{\forall \tau_j \in \theta_{k,j} \bigcup \Theta_{k,j}} (\omega^{GR}_j); \text{ if } m = 2 \end{cases} \tag{3.39}$$

and integrated in (3.38) depending on the investigated multi-core setup.

4.3. Local Blocking Time due to Cooperative Tasks. If a *lpl* task τ_j is scheduled cooperatively with τ_i, a job J_k of task τ_k may use the RESCHEDULE interface at specific scheduling points, i.e. at runnables borders (see Section 3.9.1). Thus, a job J_k of a cooperative task $\tau_k \in lplC(i)$ can block task τ_i only for the length of the non-preemptive section (i.e. often for the length of one runnable denoted here with C_{r_k}) plus the time the job J_k is directly blocked by remote tasks during the time interval C_{r_k}.

To provide a conservative upper bound on the local blocking time due to cooperative *lpl* tasks the maximum length of non-preemptive sections inside C_k, i.e. the length C_{r_k} of one runnable, plus the maximum blocking time during the time interval C_{r_k} has to be considered as follows:

$$LB^C_i(w_i(q)) = \max_{\forall \tau_k \in \text{lplC(i)}} \{(C_{r_k} + DB_k(C_{r_k}))\} \tag{3.40}$$

where $DB_j(C_{r_k})$ is given by (3.39) with the observation that the number of shared resource accesses that are issued by J_k are limited to one runnable, i.e. $n^G_{r_k}$ (remember that according to the system model in Section 3.9.1 $n^G_k = k \cdot n^G_{r_k}$).

Since on any core only one task can execute at a time, only one of the three blocking scenarios due to lower priority local tasks can actually occur. Therefore the maximum possible impact of *lpl* tasks is given by the maximum value of the three blocking time values in (3.37), (3.38) and (3.40). Thus, the **local blocking time** $LB_i(w_i(q))$ of a task τ_i due to any of the *lpl* tasks is given by:

$$LB_i(w_i(q)) = \max\left\{LB^P_i, LB^{NP}_i, LB^C_i\right\} \tag{3.41}$$

5. Overall Blocking Time in AUTOSAR conform Multi-Core ECUs. The worst-case blocking time $BT_i(w_i(q))$ that a task τ_i can encounter in a time window $w_i(q)$

is given by the sum of the **direct blocking time** (given by (3.31) in case of $m > 2$ or by (3.32) in case $m = 2$) the **indirect blocking time** (given by (3.33) in case of $m > 2$ or by (3.34) in case $m = 2$), the **blocking time due to cancelled and re-initiated requests** (given by (3.35) in case of $m > 2$ or by (3.36) in case $m = 2$) and the **local blocking time** $LB_i(w_i(q))$ in (3.41):

$$BT_i(w_i(q)) = \begin{cases} DB_i^{MC}(w_i(q)) + IB_i^{MC}(w_i(q)) + CRB_i^{MC}(w_i(q)) + LB_i(w_i(q)); \ m > 2 \\ DB_i^{DC}(w_i(q)) + IB_i^{DC}(w_i(q)) + CRB_i^{DC}(w_i(q)) + LB_i(w_i(q)); \ m = 2 \end{cases} \tag{3.42}$$

3.9.3 Response Time Analysis for Partitioned AUTOSAR Scheduling

In this section, we introduce a response-time analysis approach for tasks with arbitrary activations and deadlines which share resources in AUTOSAR conform partitioned multi-core systems. For this, we rely on the background provided by the analysis approaches for multi-core preemptive and non-preemptive systems presented in Section 3.7.2 and 3.8.2, respectively. The response-time analysis solutions for partitioned preemptive and non-preemptive multi-core setups rely on single-core processor theory, which however can not be directly applied. More exactly, two elements, the critical instant scenario (exemplified in Figure 3.8 in Section 3.5.1) and therewith the computation of the maximum level-i busy window, on which the classical response-time analysis procedure rely, have to be revisited in case of multi-core systems with shared resources.

3.9.3.1 Critical Instant in AUTOSAR conform Multi-Core Systems

It is known that the use of global shared resources may lead to the suspension of tasks which possibly defers the execution time of the tasks and thus counters the assumptions regarding the critical instant scenario on which the classical response time analysis approach rely [116]. In Section 3.7.2 the critical instant scenario was revisited for multi-core setups where shared resources are arbitrated according to the MPCP [116] policy and applications are scheduled according to partitioned SPP scheduling. Correspondingly, the computation of the maximum level-i busy windows and therewith of the worst-case response times were adapted.

However, in Section 3.8.2 it was shown that the possible deferred execution of tasks identified under MPCP shared resource arbitration and partitioned SPP scheduling does not occur in case of AUTOSAR spinlock-based resource arbitration and partitioned SPNP scheduling. This is actually ensured alone by the AUTOSAR shared resource arbitration strategy which imposes that any job which has an outstanding request for a shared resource is actively waiting for that resource without suspending itself. This means that a task which is spinning does not allow lower priority local tasks execute, fact that avoids the deferred execution issue [116].

Therefore, in case of AUTOSAR spinlock-based arbitration the classical critical instant scenario remains valid. However, in comparison to the previous approaches that address either preemptive or non-preemptive scheduling, the critical instant scenario

under AUTOSAR OS depends on the different types of tasks, i.e. preemptive, non-preemptive and cooperative, as follows:

Definition 3.2 *A task τ_i on an AUTOSAR conform multi-core system with shared resources experiences the critical instant scenario, which leads to the worst-case response time, when it is released:*

1. *at the time moment just after a job J_j of a lower priority local task $\tau_j \in lpl(i)$ has started its local execution and where J_j is the job that maximally delays the execution of τ_i through either its non-preemptable execution or the critical sections executed at a higher priority than τ_i, i.e. according to the local blocking time factor in (3.41)*

   and

2. *simultaneously with jobs of all higher priority local tasks in $hpl(i)$.*

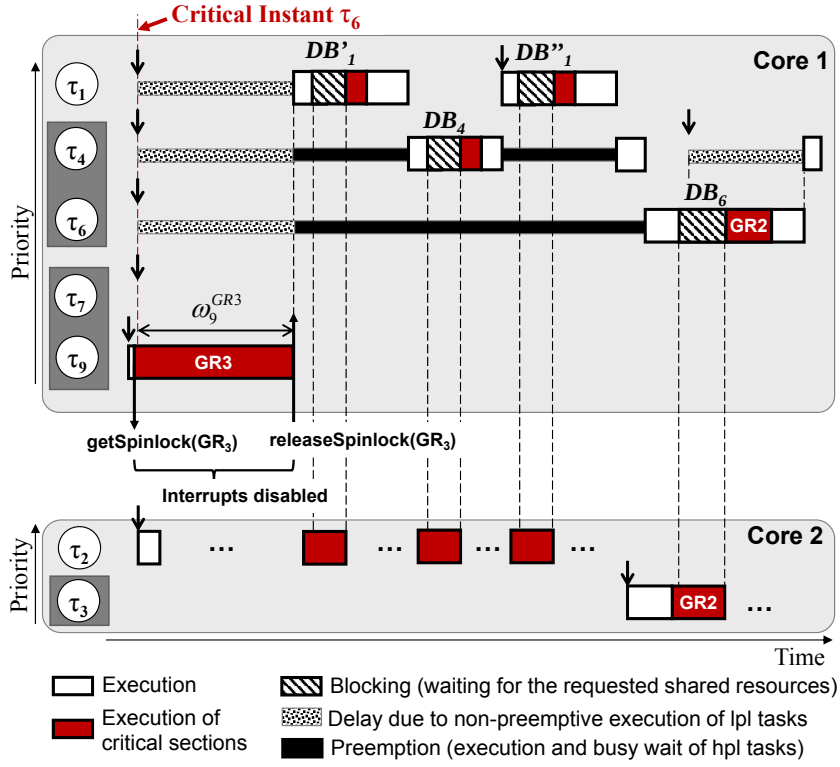


Figure 3.17: Critical instant example for task τ_6 in the multi-core system in Figure 3.15.

An example of a critical instant for task τ_6 in the multi-core system in Figure 3.15 is represented in Figure 3.17. There, task τ_6 is activated at the same moment with the higher priority tasks τ_1 and τ_4 just after the lower priority task τ_9 started its execution and locked the global resource $GR3$. According to the AUTOSAR specification (see

Section 3.9.2) τ_9 will disable all interrupts for the time it holds the global resource $GR3$ and thus it delays (blocks) the execution of all other tasks on Core 1. After τ_9 releases $GR3$ the higher priority tasks τ_1 and τ_4 can execute and their execution represents preemption time for τ_6. The terms DB in Figure 3.17 represent the direct blocking times that different jobs running on Core 1 may experience when the requested GRs are locked by jobs of the remote tasks. Thus, the direct blocking times of tasks τ_1 and τ_4 prolong the preemption time of task τ_6. Finally, when τ_6 starts executing it will run until completion without any preemption by the *hpl* task τ_4 (as this is in the same non-preemptively scheduled group with τ_6) even if τ_6 will experience direct blocking through remote tasks.

This example clearly highlights the individual influence of the core local scheduling policy and of the shared resource arbitration policy. According to the spinlock-based arbitration mechanism task τ_4 could preempt the lower priority task τ_6 when this performs busy waiting. However, the non-preemptive execution of τ_6 against τ_4 is enforced through their membership to the same (non-preemptive) group of tasks.

3.9.3.2 Derivation of the Maximum Level-i Busy Window

Similar to the analysis approaches in Section 3.7.2 and 3.8.2 the worst-case response time of a task τ_i on an AUTOSAR conform multi-core system with shared resources is given by the largest response time of any of the q ($q = 1 \ldots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the maximum level-i busy window.

Assuming the critical instant scenario under partitioned AUTOSAR scheduling in a multi-core setup, the maximum level-i busy window of a task τ_i consists not only of the time intervals during which the tasks contributing to the busy window execute but also of the time intervals these tasks are blocked and have to wait for the required GRs (see Figure 3.17). According to the blocking time analysis introduced in Section 3.9.2 the blocking time of a task in multi-core systems is a function of the window size during which the task initiates requests to the required shared resources. Thus, the maximum level-i busy window L_i of a task τ_i in an AUTOSAR conform multi-core system can be calculated with the following recurrence relation:

$$L_i^{n+1} = BT_i(L_i^n) + \eta_i^+(L_i^n) \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(L_i^n) \cdot C_j \tag{3.43}$$

where $BT_i(L_i^n)$ represents the blocking time of task τ_i in the busy window L_i^n given by (3.42); $\eta_i^+(L_i^n) \cdot C_i$ represents the maximum workload of task τ_i in the busy window L_i^n and $\eta_j^+(L_i^n) \cdot C_j$ represents the maximum workload of the tasks with higher priority than τ_i in the busy window L_i^n.

Similar to (3.17) in Section 3.7.2 and (3.26) in Section 3.8.2 all components of (3.43) grow monotonically with respect to the window size. This allows the iterative calculation of a solution. The recurrence relation (3.43) starts with an initial value $L_i^0 = C_i$, and finishes when $L_i^{n+1} = L_i^n$. (i.e. two consecutive iterations provide identical results). The recurrence relation is guaranteed to converge if the resource utilization is less than 100%. Because in multi-core setups with spinlock-based shared resource arbitration the

utilization of each individual core is a function not only of the tasks' core execution times but also of the blocking times, the iterative calculation has to be stopped if the "effective" core utilization level (composed of the tasks' core execution times and the spinning times) exceeds 100% at some iteration point. In that case the task set is considered unschedulable.

3.9.3.3 Derivation of the Worst-Case Response Times

To determine the WCRT of any task τ_i, it is necessary to calculate the response time for all jobs which occur in the maximum level-i busy window, i.e. for each task instance q ($q = 1 \dots Q_i, Q_i \in \mathbb{N}^+$) and $Q_i = \eta_i^+(L_i)$) with L_i obtained with (3.43).

In comparison to previous work, which independently handled one scheduling policy at once (i.e. either SPP or SPNP), the response time derivation in case of partitioned AUTOSAR multi-core scheduling has to consider the different types of tasks that can execute on a core. In what follows we introduce the equations for the response-time analysis that covers all possible types of scheduling, i.e. preemptive, non-preemptive and mixed-preemptive scheduling (see Section 3.9.1). Response time equations for fully preemptive and non-preemptive partitioned multi-core scheduling were introduced in Section 3.7.2 and 3.8.2 but will be briefly refined here in order to introduce specific response time equations for the more complex case of mixed-preemptive scheduling under partitioned AUTOSAR scheduling and spinlock-based shared resource arbitration.

1. Response-time procedure for preemptable tasks.

For any task τ_i that is fully preemptable, i.e. is not part of any group of tasks (e.g. τ_1, τ_2 and τ_{10} in Figure 3.15), or if τ_i is the highest priority task in any group of tasks (e.g. τ_3, τ_4 and τ_7 in Figure 3.15) the WCRT is given by the largest response time of any of the q ($q = 1, \dots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the maximum busy window $w_i(q)$ as follows (see also Section 3.7.2)

$$R_i = \max_{q=1 \dots Q_i} (w_i(q) - \delta_i^-(q)) \tag{3.44}$$

where the busy window $w_i(q)$ of the q-th activation is obtained by iteratively solving:

$$\begin{aligned} w_i^{n+1}(q) =& q \cdot C_i + BT_i(w_i^n(q)) \\ &+ \sum_{\forall \tau_j \in hplP(i)} \eta_j^+(w_i^n(q)) \cdot C_j \end{aligned}$$

which can be rewritten as

$$\begin{aligned} w_i^{n+1}(q) =& q \cdot C_i + LB_i(w_i^n(q)) + DB_i(w_i^n(q)) + CRB_i(w_i^n(q)) \\ &+ \sum_{\forall \tau_j \in hplP(i)} (\eta_j^+(w_i^n(q)) \cdot C_j + DB_j(w_i^n(q))) \end{aligned} \tag{3.45}$$

Depending on the number of processor cores in the system, the direct blocking terms DB are given by (3.31) or (3.32) and the blocking term CRB by (3.35) or (3.36). The maximum workload due to higher priority local tasks is prolonged by the time these tasks can be directly blocked by remote tasks. Capturing this effect with the sum over all tasks in $hplP(i)$ the indirect blocking time of task τ_i is implicitly considered. Furthermore, the local blocking time, the direct blocking time and the blocking time due to re-initiated cancelled requests of τ_i are added.

2. Response-time procedure for non-preemptable tasks. A task τ_i is fully non-preemptable in case all tasks on τ_i's host core are part of the same group of non-cooperatively scheduled tasks or if τ_i is part of a non-preemptable group and there is no other task with higher priority than τ_i which is not in τ_i's group. These setups are not covered in the system example in Figure 3.15, however, examples of fully non-preemptable tasks would be τ_3 and τ_4 if τ_1 and τ_2 would not be mapped on Core 1 and Core 2 or if they would be in the same group with τ_3 and τ_4.

The WCRT of a fully non-preemptable tasks is given by the largest response time of any of the q ($q = 1, \dots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the busy window $w_i(q)$ as follows (see also Section 3.8.2):

$$R_i = \max_{q=1\dots Q_i} \left(w_i(q) + C_i - \delta_i^-(q) \right) \tag{3.46}$$

where the busy window $w_i(q)$ of the $(q-1)$-th activation (i.e the queueing delay of the q-th activation) is computed with:

$$\begin{aligned} w_i^{n+1}(q) =& (q-1) \cdot C_i + LB_i(w_i^n(q)) + DB_i(w_i^n(q)) \\ &+ \sum_{\forall \tau_j \in hplNP(i)} \left(\eta_j^+(w_i^n(q)) \cdot C_j + DB_j(w_i^n(q)) \right) \end{aligned} \tag{3.47}$$

The terms in (3.46) and (3.47) are similar to the ones introduced above for the analysis of fully preemptable tasks, with the difference that the considered higher priority tasks are in the set $hplNP$.

The main difference when handling preemptive and non-preemptive tasks is given by the way the terms $w_i(q)$ and R_i are calculated. In case of fully preemptable tasks $w_i(q)$ represents the level-i busy window, whereas in case of non-preemptable tasks $w_i(q)$ represents the queueing delay. As already known from the single-processor and multiprocessor scheduling theory [42, 88] in order to obtain the response time of a task τ_i under non-preemptive scheduling the core execution time C_i has to be added to the queueing delay $w_i(q)$ - see (3.46) vs. (3.44). Furthermore, in case all tasks on a processor core are non-preemptive, the blocking time CRB_i due to re-initiated cancelled requests for global resources is 0 and therefore omitted here - see (3.47) vs. (3.45).

3. Response-time procedure for mixed-preemptable tasks. In case of mixed-preemptive tasks one has to jointly consider the maximum workload caused by the higher

priority local tasks to which τ_i is both preemptable and non-preemptable. Furthermore, one has to handle the cases where tasks in a group are non-preemptive or cooperative.

3.1 Tasks in a group are non-preemptable to each other. For the case tasks in a group are configured as fully non-preemptable, the response time R_i is computed with

$$R_i = \max_{q=1...Q_i} (w_i(q) + C_i - \delta_i^-(q))$$

which is the same as (3.46). However, for the computation of $w_i(q)$ for the q-th activation of a mixed-preemptable task τ_i we refine the equations above to cover the different types of higher priority tasks as follows:

$$\begin{aligned} w_i^{n+1}(q) =& (q-1) \cdot C_i \\ & + LB_i(w_i^n(q) + C_i) + DB_i(w_i^n(q) + C_i) + CRB_i(w_i^n(q) + C_i) \\ & + \sum_{\forall \tau_j \in hplP(i)} (\eta_j^+(w_i^n(q) + C_i) \cdot C_j + DB_j(w_i^n(q) + C_i)) \\ & + \sum_{\forall \tau_j \in hplNP(i)} (\eta_j^+(w_i^n(q)) \cdot C_j + DB_j(w_i^n(q))) \end{aligned} \tag{3.48}$$

The key idea when computing $w_i(q)$ for mixed-preemptable tasks is to differentiate between the time intervals where task τ_i can be preempted and where not. Therefore, the clauses in (3.48) compute: (i) the queueing delay of the q-th activation of task τ_i due to its previously executed instances; (ii) the local blocking time, the direct blocking time and the blocking time due to re-initiated cancelled requests in a time interval $w_i(q)+C_i$, i.e. we consider not only the blocking of the $q-1$ activations but also the blocking the q-th activation will experience; (iii) the interference all activations, including the analyzed one q, of task τ_i will experience due to the higher priority tasks that can preempt τ_i; (iv) the interference all activations, up to the analyzed one, of task τ_i will experience due to the higher priority tasks that cannot preempt τ_i. Similar to the classical SPNP scheduling analysis (see also 3.9.3.3 - 2 above) the execution of the q-th activation is considered in the response-time equation (3.46).

3.2 Tasks in a group are cooperative to each other. For the case tasks in a group are configured as cooperative, the analysis procedure is similar to the one for fully non-preemptable tasks inside a group, with the difference that instead of handling the jobs J_i of the tasks in a group one has to consider the execution of their runnables r_i. More exactly, this means that for each activation q of a job J_i composed of K_i *identical runnables* r_i of size C_{r_i} we have $q' = q * K_i$ runnable instances. In other words, as all runnables of a tasks are identical for each activation q of a job J_i one can consider $q' = q*K_i$ activations of a runnable. Inside a group the execution of runnables belonging to different jobs are scheduled according to the SPNP scheduling. Thus, with the busy window and response time equations we have to capture:

(i) *the scheduling of runnables according to the group internal policy*

(ii) *the scheduling of jobs of higher priority tasks that are not in the group.*

Therefore, the worst-case response time $R'_i(q')$ of any of the q' mixed-preemptable runnables r_i of a task τ_i is given by

$$R'_i(q') = w'_i(q') + C_{r_i} - \delta_i^-(\lceil \frac{q'}{K_i} \rceil), \forall q' = 1 \dots K_i * Q_i \tag{3.49}$$

where $\delta_i^-(\lceil \frac{q'}{K_i} \rceil)$ actually captures the minimum distance relative to the activation q of the analyzed task τ_i, i.e. $\delta_i^-(\lceil \frac{q'}{K_i} \rceil) = \delta^-(q)$, and $w'_i(q')$ for the q'-th activation of a mixed-preemptable runnable r_i is computed with

$$\begin{aligned} {w'_i}^{n+1}(q') =& (q'-1) \cdot C_{r_i} \\ & + LB_i({w'_i}^n(q') + C_{r_i}) + DB_i({w'_i}^n(q') + C_{r_i}) + CRB_i({w'_i}^n(q') + C_{r_i}) \\ & + \sum_{\forall \tau_j \in hplP(i)} (\eta_j^+({w'_i}^n(q') + C_{r_i}) \cdot C_j + DB_j({w'_i}^n(q') + C_{r_i})) \\ & + \sum_{\forall \tau_j \in hplC(i)} (\eta_j^+({w'_i}^n(q')) \cdot K_j \cdot C_{r_j} + DB_j({w'_i}^n(q'))) \end{aligned} \tag{3.50}$$

The key idea when computing $w'_i(q')$ for mixed-preemptable runnables is to differentiate between the time intervals where a runnable r_i of a task τ_i can be preempted and where not. Therefore, the clauses in (3.50) compute: (i) the queueing delay of the q'-th activation of runnable r_i due to its previously executed runnable instances; (ii) the local and direct blocking time and the blocking time due to re-initiated cancelled requests in a time interval $w'_i(q') + C_{r_i}$, i.e. we consider not only the blocking of the $q'-1$ activations but also the blocking the q'-th activation will experience; (iii) the interference all runnables r_i, including the analyzed one q', will experience due to the higher priority tasks that can preempt τ_i. Tasks $\tau_j \in hplP(i)$ that are not in the same group with τ_i may preempt each runnable r_i; (iv) the interference all runnables r_i, up to the analyzed one, will experience due to the higher priority runnables, i.e. runnables r_j of tasks $\tau_j \in hplC(i)$ that cannot preempt but delay runnables r_i of τ_i. Similar to the classic SPNP scheduling analysis (see also 3.9.3.3 - 2 and 3.9.3.3 - 3.1 above) the execution of the q'-th runnable activation is considered in the response-time equation (3.49).

With equation (3.49) and (3.50) we obtain the response times for each of the q' runnable activations of any task τ_i . As we are interested in the worst-case response-time of the task τ_i, we have to consider the obtained response times for the last runnable of each job J_i, i.e. runnable K_i. As there can be Q_i jobs of the tasks τ_i in the busy window, we have to consider the response times of $K_i, 2 \cdot K_i, \dots Q_i \cdot K_i$. Thus, the worst-case response time of a cooperatively scheduled task τ_i is given by:

$$R_i = \max_{q=1 \dots Q_i} R'_i(q \cdot K_i)) = \max_{q'=1 \dots K_i * Q_i} R'_i(q')) \tag{3.51}$$

with R'_i obtained with (3.49).

Finally, if worst-case response time values R_i have been obtained for all the tasks in the multi-core system, with (3.44) for preemptable tasks, with (3.46) for non-preemptable

tasks and with (3.48) and (3.51) in case of mixed-preemptable tasks, the schedulability test consists of checking whether the condition $R_i \leq D_i$ holds for every task τ_i.

3.10 System-Level Analysis Integration

Section 2.3.1 introduced the general compositional system-level analysis procedure for multi-core systems with shared resources and Section 2.3.2 established general conditions for this to converge towards a fixed-point. Next, we address the integration of the blocking- and response-time analyses, introduced across the previous sections of this chapter, in the system-level analysis procedure and show that all analysis elements fulfill the conditions of Corollary 2.2.

From Section 2.3.1 we know that the compositional system-level analysis procedure consists of an iterative analysis flow (i) in which separate local component analyses (in our case response-time analysis based on the busy-window technique per core and bus) are interleaved with the propagation of event models and (ii) which is repeated until a system-wide convergence. Furthermore, we know (see Section 2.3.1 and Section 3.5.2) that in order to derive timing bounds of multi-core applications which share secondary resources the local timing analysis procedures are extended with additional steps, namely the shared resource load derivation, the blocking time analysis and finally the integration of the derived blocking times in the worst-case response times (see also Figure 2.5).

The analysis elements introduced in this chapter for computing worst-case timing bounds of partitioned multi-core setups are integrated in the compositional system-level iterative analysis flow as follows:

1. Given a set of task activating event models η^+ for each task in the system, a set of shared resource access event models $\tilde{\eta}^+$ is derived with equation (3.8) or (3.9).
2. Based on the shared resource access event models, the shared resource access delays (i.e. the blocking times) are calculated for each task depending on the arbitration strategy with:
   - equation (3.16) in Section 3.7.1 for the MPCP arbitration strategy under SPP core local scheduling;
   - equation (3.23) in Section 3.8.1 for the MLP-NP arbitration strategy under SPNP core local scheduling;
   - equation (3.42) in Section 3.9.2 for the AUTOSAR spinlock-based resource arbitration strategy under AUTOSAR conform core local scheduling.
3. The respective blocking times then become part of the response time analysis of each task on each core, following:
   - the equations in Section 3.7.2 for partitioned multi-core SPP scheduling;
   - the equations in Section 3.8.2 (especially in 3.8.2.3) for partitioned multi-core SPNP scheduling;
   - the equations in Section 3.9.3 (especially in 3.9.3.3) for partitioned AUTOSAR conform multi-core scheduling.

Based on the obtained response time values updated output event models η' (see Figure 2.5) can be derived e.g. by converting the results of (2.6) and (2.7) [24].

The process is repeated as long as any event model estimate has been refined. This procedure is conceived to be appropriate not only for time-triggered task activations but also for event driven task activations. In multi-core and multiprocessor systems tasks' activations may be the result of other tasks finishing or data arriving over a bus such that tasks' activating event models may not initially. In such cases the compositional analysis approach starts with initial estimates (i.e. starting point generation) which are refined through iteration [119, 121, 64].

However, the system-level analysis procedure faces additional challenges when applied to multi-core systems with shared resources. These are given by the various mutual dependencies between the task activating event models η^+, the shared resource delays $\tilde{\eta}^+$, and the task response times. More exactly, the response time R_i of a task τ_i on a core is a function of the activating event model η_i. But, as can be observed from the various equations in the previous sections - e.g. (3.43), (3.45), (3.47), (3.48), (3.50) and (3.8) - and as illustrated in Figure 3.18 - the length of the busy windows and the tasks' response times in multi-core systems with shared resources depend also on the delay caused by the use of shared resources, i.e. the response time R_i is a function of blocking time B_i. This delay in turn depends on the amount of traffic imposed on the shared resources by other tasks on other processors $\tilde{\eta}_j$ (as can be seen in the blocking time equations in Sections 3.7.1, 3.8.1 and 3.9.2), which in turn is a function of the respective local load η_j. This translates into a dependency cycle between the local response time analysis on the different cores, challenging the entire response time analysis procedure for multi-core systems.

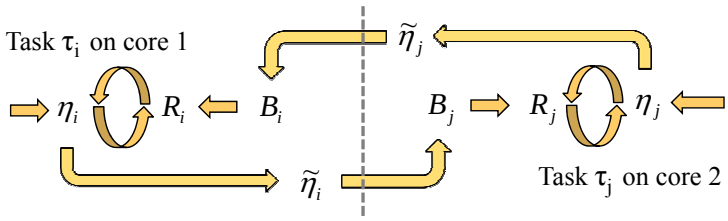


Figure 3.18: Dependencies in the response-time analysis procedure.

In order to avoid these dependencies, the shared resource request bound in (3.8) can be replaced by the bound in (3.9) which is independent of the task's response time, or it can be computed through iteration (started with an initial value $R_j = 0$, $\forall \tau_j$ mapped on remote cores) as long as all analysis parameters are monotonic (or their sets form a complete partial order CPO - see Section 2.3.2).

Furthermore, the response times of tasks in a multi-core system with voluntary suspension (as handled in Section 3.7.2) can not be calculated in an arbitrary sequence, because

[24] Remember that the functions η and δ are pseudo-inverse to each other (see Figure 2.1).

(3.19) requires the knowledge of the response times of higher priority local tasks. To tackle this dependency the response times on each core can be calculated top-down, starting with the highest-priority task.

The iterative system-level analysis procedure represents a fixed-point problem, which can be solved only if the conditions of Corollary 2.2 are fulfilled for each local analysis procedure and each analysis parameter. The conditions demand that the analysis functions are order preserving with respect to their input parameters and that the set of the analysis results forms a complete partial order.

Order Preservation on Complete Partially Ordered Sets.

The building blocks of the system-level analysis procedure are the local response-time analyses (for SPP, SPNP and AUTOSAR conform scheduling) based on the busy window approach [154]. Thus, the response-time and the busy window analysis functions for the different scheduling policies considered in this thesis represent the central elements of the system-level approach and must adhere to the conditions of Corollary 2.2.

Depending on the scheduling policy the response time R_i of a task τ_i and the maximum busy window $w_i(q)$ of q activations of τ_i are given by equations:

- (3.18) and (3.19) for static-priority preemptive scheduling and suspension based shared resource arbitration;
- (3.27) and (3.28) for static-priority non-preemptive scheduling and spinlock-based shared resource arbitration;
- (3.44) and (3.45) for AUTOSAR conform preemptive scheduling, (3.46) and (3.47) for AUTOSAR conform non-preemptive scheduling and (3.46), (3.48), (3.49) and (3.50) for AUTOSAR conform mixed-preemptive scheduling, all these under AUTOSAR conform spinlock-based shared resource arbitration.

In what follows, we won't address all these equations, but exemplary focus on the analysis equations for SPP scheduling and suspension based blocking according to the MPCP algorithm in Section 3.7.2. Due to similarities between the analysis procedures the general argumentation provided next applies also for the other algorithms provided in this chapter.

Theorem 3.1 *The response-time analysis and the busy window analysis of tasks scheduled under partitioned multiprocessor static-priority preemptive scheduling which share secondary resources according to the MPCP strategy is order preserving.*

Proof: We have to show that for each analysis state achieved by iteration the response time analysis delivers increasing worst-case response time values. More exactly, we have to show that for two successive parametrizations j and $j+1$ of the event model EM_i associated to task τ_i (see Definition 2.7 and (2.9) and (2.10)), i.e. for the event model estimate EM_i^j of task τ_i in the analysis state as_j and the event model estimate EM_i^{j+1} of task τ_i in a successive analysis state as_{j+1} we have:

$$EM_i^j \leq EM_i^{j+1} \Rightarrow R_i(EM_i^j) \leq R_i(EM_i^{j+1})$$

Under static priority preemptive scheduling and MPCP conform suspension based blocking, the worst-case response time of a task τ_i is given by the largest response time of any of the q ($q = 1 \dots Q_i, Q_i \in \mathbb{N}^+$) task activations that lie within the maximum busy window $w_i(q)$ according to (3.18) which is:

$$R_i = \max_{q=1\dots Q_i} (w_i(q) - \delta_i^-(q)) \qquad (1)$$

and (3.19) which is:

$$w_i(q) = q \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i(q) + R_j) \cdot C_j + BT_i(w_i(q)) \qquad (2)$$

The relation $\leq$ between two event model estimates have to be read with "more generic", which means "no less events in any time interval" [130]. Formally, an event model EM_i^j of a task τ_i is more generic than another EM_i^{j+1}, if

$$EM_i^j \leq EM_i^{j+1} \Rightarrow \\ \forall q : \delta_i^{j,-}(q) \geq \delta_i^{j+1,-}(q) \Rightarrow \forall\, \Delta t \geq 0 : \eta_i^{j,+}(\Delta t) \leq \eta_i^{+,j+1}(\Delta t) \qquad (3)$$

which means that whereas the minimum distances between any q task activations may only decrease or remain unchanged the maximum number of task activations may only increase or remain unchanged. The order on event model estimates was proved in Chapter 3 in [142].

From (3) we know that $\delta_i^-(q)$ in relation (1) above may only decrease or remain unchanged, thus, in order for the response time function (1) to be order preserving we need to prove that the busy window function in relation (2) above is order preserving.

(2) is order preserving if all its elements are order preserving with respect to the analysis states. As the addition and multiplication operators are order preserving, we need to show that each individual factor in (2) is order preserving:

- The first factor $q \cdot C_i$ captures the task own execution during the investigated time interval and is composed of the constant factor C_i and the number of considered task activations q which can only increase or remain unchanged.
- The second factor is a sum over all higher priority tasks mapped on the same resource as τ_i which considers the function η^+ and the constant factor C_j. The function η_j^+, which return the maximum number of events in a time interval, remains order preserving if the response time R_j of the higher-priority tasks is order preserving. As the response times on each local analysis are computed top down, for the tasks with the highest priority the term R_j is omitted and for the rest R_j will be order preserving if the response time function for the previously analyzed tasks is order preserving.

- The third factor $BT_i(w_i(q))$ in (2) corresponds to (3.16), each blocking term of (3.16) being a function of the load $\tilde{\eta}_j^+(w_i(q))$ imposed by other tasks τ_j in the system on the shared resources and of other parameters. These parameters are either constant, such as the size of the critical sections ω_j^{LR}, ω_j^{GR} or the number of shared resource accesses per task instance n_i^G, or are order preserving such that the number of considered task activations q. Thus, the blocking time analysis is order preserving if the shared resource request bound function $\tilde{\eta}_j^+$ is order preserving. This however, is inherent to (3.8) where a specific event model estimate η^+ is scaled by a constant factor or (3.9) where the number of issued shared resource requests increases with the size of the investigated time window, which is always divided by the constant factor d_{srr}.

As all individual factors on the right hand side of (2) are order preserving the busy window analysis function is order preserving. Therewith, all functions of the local response time analysis procedure are order preserving and all their input parameters are either constant or become more generic with each iteration, i.e. form a complete partial order set. Theorem 3.1 follows. □

As all elements (i.e. functions and parameters) of the analysis procedures for SPNP scheduling and AUTOSAR conform scheduling are similar to those of the analysis for SPP scheduling handled above, the argumentation in Theorem 3.1 holds for all of them.

Corollary 3.2 *The response-time analyses and the busy window analyses of tasks scheduled under static-priority preemptive, static-priority non-preemptive and mixed-preemptive scheduling as introduced in Section 3.7.2, 3.8.2 and 3.9.3 are order preserving and the set of each input parameter forms a complete partial order.*

Based on this knowledge we can conclude that the two conditions of Corollary 2.2 are fulfilled for all components of the system-level analysis procedure (i.e. for the local analysis functions) and therewith for the global analysis function itself (according to Corollary 2.1).

Given the order-preservingness of the extended system-level analysis procedure the analysis will either converge (i.e. all task activating event models η^+ and all shared resource request bounds $\tilde{\eta}^+$ have not changed after an iteration and lead to identical response-time analysis results) towards a fixed point, which represent a conservative solution, or the event model estimates grow to infinity, in which case the analysis will be stopped as soon as a real-time constraint (e.g. deadline of a task) is violated.

3.11 Experimental evaluation

In this section we present the evaluation of the analysis approaches introduced in the previous sections of this chapter and show their applicability to different multi-core use-cases. For evaluation we mainly consider the multi-core setup in Figure 3.1b).

3.11.1 Evaluation of Multi-Core Setups under Partitioned SPP Scheduling and MPCP Shared Resource Arbitration

3.11.1.1 Benefits of using an enhanced Model for the Shared Resource Load Derivation

In a first experiment we compare our response-time analysis method introduced in Section 3.7 with the analysis presented in [116]. As the cited method is only applicable to periodic systems, we assume that all tasks in the system in Figure 3.1 are stimulated periodically with the parameters given in Table 3.2.

Table 3.2: Particular configuration of the parameters for the system in Figure 3.1b under partitioned SPP scheduling and MPCP shared resource arbitration.

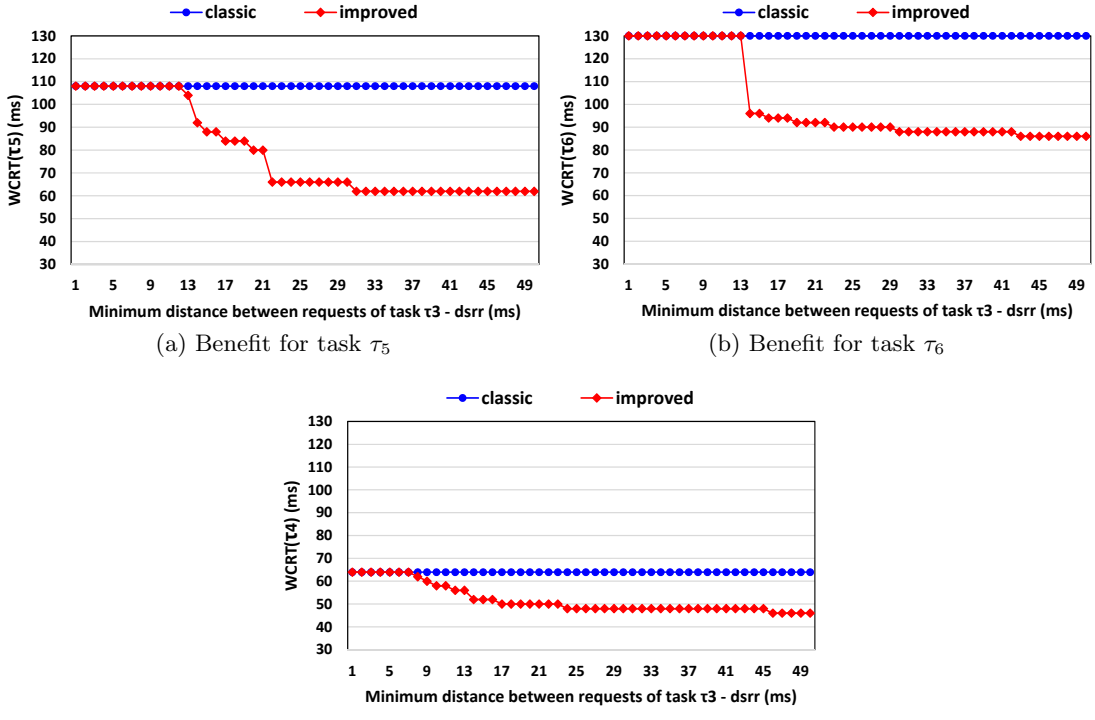
Mapping	Task Name	Event Stream	Priority	Period T_i (ms)	Core Execution Time C_i (ms)	Global Resource Accesses $n_i^G * \omega_i^G$	Local Resource Accesses $n_i^L * \omega_i^L$
Core 1	τ_1	η_1	1	500	[30,30]	1 * 2 to GR1	1 * 2 to LR1
Core 1	τ_3	η_3	3	1000	[500,500]	9 * 2 to GR2	1 * 2 to LR1
Core 2	τ_4	η_4	4	75	[30,30]	2 * 2 to GR1	1 * 2 to LR2
Core 2	τ_6	η_6	6	150	[10,10]	1 * 2 to GR2	1 * 2 to LR2
Core 3	τ_2	η_2	2	90	[10,10]	1 * 2 to GR2	1 * 2 to LR3
Core 3	τ_5	η_5	5	1000	[20,20]	3 * 2 to GR2 1 * 2 to GR1	1 * 2 to LR3

For this particular setup, which was manually determined, both analysis approaches deliver the same worst-case response times for all tasks in the system, e.g. as illustrated in Figure 3.19 $WCRT(\tau_4) = 64ms$, $WCRT(\tau_5) = 108ms$ and $WCRT(\tau_6) = 130ms$.

This is not the case anymore when we take advantage of the improved shared resource models. To investigate the benefit, we assume that task τ_3 performs some local computation between the shared resource requests. Thus, the shared resource request bound of τ_3 considers that requests initiated by τ_3 for shared resources, which represent direct remote blocking of tasks τ_5 and τ_6, are separated by a minimum distance of d_{srr}. This is captured by the function $\tilde{\eta}_3(\Delta t) = \lceil \Delta t / d_{srr} \rceil$ (see (3.9)). The larger this distance becomes, the lower is the load imposed on the shared resource. Figure 3.19a and 3.19b show that the reduced load allows task τ_5 and τ_6 on the other processor to finish faster, because less shared resource accesses by τ_3 can fall into the response time of task τ_5 and τ_6. Indirectly, the faster execution draws less local interference on the individual cores, causing an additional benefit not only for the response time of task τ_5 and τ_6 but, as illustrated in Figure 3.19c, also for the response time of τ_4. With increasing request distances, the benefit of using our approach increases, being for $d_{srr} = 46$ around 43% more accurate in case of task τ_5, 34% in case of task τ_6 and 29% in case of task τ_4.

3.11.1.2 Response-Time Analysis applied to randomly generated Multi-Core Setups

In this set of experiments we demonstrate the applicability of the approach presented in Section 3.7 by analysing the timing behavior of a set of pseudo-randomly generated test cases for the multi-core system in Figure 3.1b.



(a) Benefit for task τ_5

(b) Benefit for task τ_6

(c) Benefit for task τ_4

Figure 3.19: Benefit of using the minimum distance between requests d_{srr} in the shared resource request derivation on the tasks' worst-case response times:
- "classic" - response times obtained with the analysis in [116]
- "improved" - response times obtained with the new analysis in Section 3.7.

Basic Configuration Parameters.

The activation period, the activation jitter, and the worst-case execution time (WCET) per task are generated according to the UUnifast algorithm [19] as follows: the utilization on each core is assumed to be $U_{core} = 50\%$; based on the assumed core utilization each task on a core is assigned a random utilization U_i such that the sum of all task utilizations on that core equals the total core utilization ($\forall \tau_i$ mapped on $core : \sum U_i = U_{core}$); tasks' activation periods P_i are generated randomly between 100ms and 1000ms; based on the chosen periods the tasks WCETs C_i are assigned to match the task utilization U_i. Furthermore, in order to deviate from the periodic assumptions, each task can be randomly assigned an input jitter from the interval $[0, 2 \cdot P_i]$ (i.e. each task can potentially be activated by a maximum burst of 3 simultaneous activations).

The number of critical sections per task is assumed to be constant 4 for all tasks in the system. Thus, each task in the system Figure 3.1b is assumed to perform two accesses to each local shared resource and two accesses to the global shared resources as indicated in Table 3.3. Critical sections are considered not nestable. The size of each critical section is generated as follows: the total size of critical sections per task instance

Table 3.3: Accesses to the shared resources for the task in Figure 3.1b.

Task	Accesses to Global Resources $n_i^G * \omega_i^G$	Accesses to Local Resources $n_i^L * \omega_i^L$
τ_1	2 to GR1	2 to LR1
τ_3	2 to GR2	2 to LR1
τ_4	2 to GR1	2 to LR2
τ_6	2 to GR2	2 to LR2
τ_2	2 to GR2	2 to LR3
τ_5	1 to GR2 and 1 to GR1	2 to LR3

CS_{total} is generated randomly to be a percentage value $x\%$ ($x \in \mathbb{N}$) of its WCET, i.e. $CS_{total} = x\% \cdot C_i, \forall \tau_i$; then, the total size of critical sections is equally split among the maximum number of critical sections per task instance, in our case this being 4.

Evaluation.

In a first set of experiments we randomly generated system configurations for the multi-core setup as follows. In a first step the activation period and the worst-case execution time of each task were randomly generated as described above, whereas the activation jitter was always considered 0 (i.e. we generated only periodic task activations). In the second step, the total size of critical sections per task instance was iteratively varied between $x\% = 1\% \ldots 25\%$ ($x \in \mathbb{N}$) of its WCET. The length of each individual critical section of a task was obtained by equally splitting the total length of critical sections among the maximum number of critical sections per task instance. The minimum distance between requests d_{srr} in (3.9) was set up to be $d_{srr} = (C_i - \omega_i^{GR})/(n_i^G + n_i^L - 1)$ [25], i.e. accesses for shared resources are equally spread across C_i.

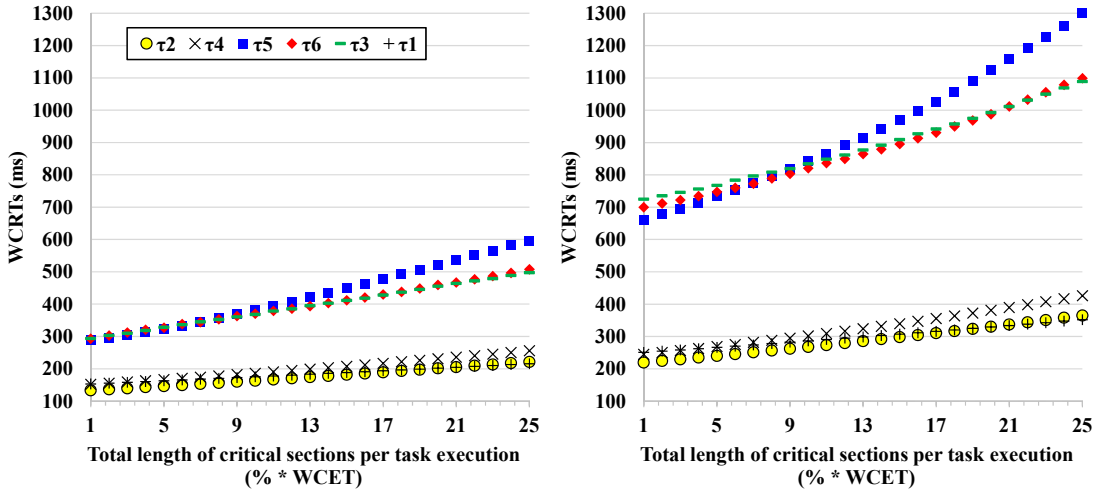
With this procedure we generated multiple test cases to which we applied the response-time analysis method for partitioned SPP scheduling and MPCP shared resource arbitration until we got 1000 schedulable configurations.

In the next set of experiments we generated and analyzed system configurations similar to the ones in the first set, with the difference that for each test case we randomly assigned an input jitter from the interval $[0, 2 \cdot P_i]$ to each task in the system.

Figure 3.20a and 3.20b depict the worst-case response times (WCRTs) depending on the critical sections length for systems with strict periodic tasks and for systems with bursty task activations. For each task the average WCRT over the 1000 setups per critical section length is given.

As expected, increasing the size of the critical sections led to increased blocking times and thus to increased response time values. From the perspective of each task, the increased critical sections length causes an increased delay not only on its own execution but also on the execution of the lower and of the higher priority local tasks. These delays are also parts of the tasks worst-case response times. As illustrated in Figure 3.20b in case of bursty task activations there is an over-proportional growth of the WCRTs. The growth is more evident for the tasks with the lowest priorities on each core, i.e. tasks τ_3, τ_5 and τ_6, these tasks being strongly affected by the bursty activations of the higher

[25] This is equivalent to the right hand size of equation (3.10). See also Figure 3.11b.



(a) Strict periodic task activations.

(b) Task activations with jitter.

Figure 3.20: Worst-case response time depending on the critical sections length for the tasks in the system Figure 3.1b) under partitioned SPP scheduling and MPCP shared resource arbitration.

priority local tasks. Depending on the tasks deadlines such an increase may eventually lead to deadline misses.

3.11.2 Evaluation of Multi-Core Setups under Partitioned SPNP Scheduling and MLP-NP Shared Resource Arbitration

3.11.2.1 Response-Time Analysis applied to randomly generated Multi-Core Setups

In this section we demonstrate the applicability of the analysis approach introduced in Section 3.8 by analyzing the timing behavior of a set of pseudo-randomly generated test cases for the multi-core system in Figure 3.1b under partitioned SPNP scheduling and MLP-NP shared resource arbitration (see Section 3.8.1.1).

Basic Configuration Parameters.

The configuration of the system parameters was performed similar to Section 3.11.1.2. The activation period, the activation jitter, and the worst-case execution time (WCET) per task were generated according to the UUnifast algorithm [19]. The utilization on each core was assumed 50%; each task on a core was assigned a random utilization that all add to 50%; the periods of the tasks were generated randomly between 100ms and 1000ms and each task was randomly assigned an input jitter from the interval $[0, 2 \cdot P_i]$; based on the chosen periods the tasks' worst-case execution times (WCETs) C_i were calculated to match their utilizations.

However, the local shared resources were assumed this time as exclusively used by the tasks mapped on the same core (this is the case under SPNP scheduling) and therefore the time that tasks spend executing local critical sections was considered part of the

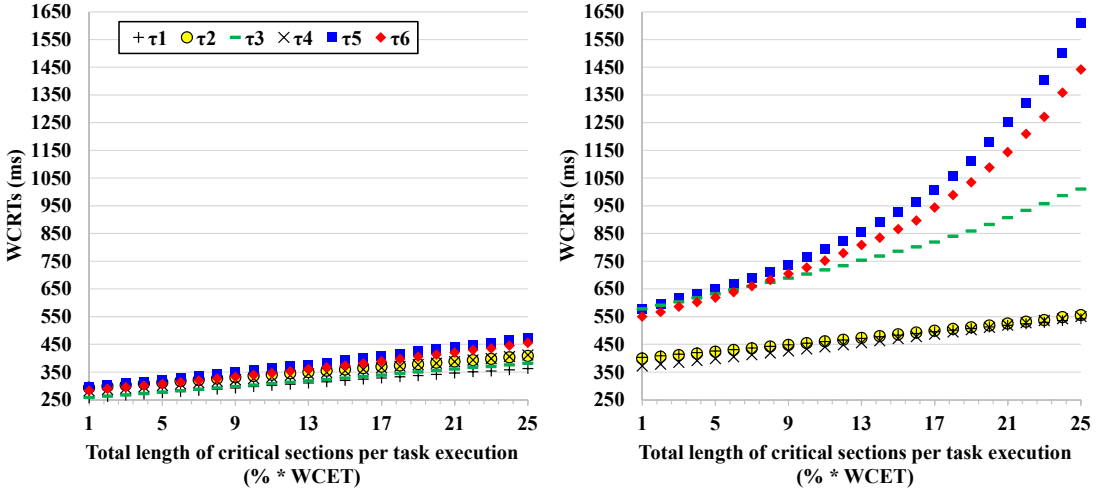
WCETs. The number of global critical sections remained constant for all tasks, being in this case 2. Therewith, we focused on the impact of the length of the global critical sections on the timing behavior of the individual cores.

The minimum distance between requests d_{srr} (see (3.9)) of each task τ_i was set up to be $d_{srr} = (C_i - \omega_i^{global})$, i.e. tasks access the global shared resources at the beginning and at the end of their WCETs C_i.

Evaluation.

We applied the response-time analysis method for partitioned SPNP scheduling and MLP-NP shared resource arbitration to multiple test cases until we got 1000 schedulable configurations for two setups namely, a) where tasks are activated strictly periodically and b) for the case that tasks activations can experience a jitter. For each generated test case the total length of the global critical sections per task instance was varied iteratively from 1% to 25% of the WCET.

Figure 3.21a) and b) depicts the worst-case response times depending on the global critical sections' length for systems with strict periodic task activations and for systems with bursty task activations. For each task the average WCRT over the 1000 setups per critical section length is given.



(a) Strict periodic task activations.

(b) Task activations with jitter.

Figure 3.21: Worst-case response time depending on the critical sections length for the tasks in the system Figure 3.1b) under partitioned SPNP scheduling and MLP-NP shared resource arbitration.

As can be seen in the diagrams of Figure 3.21, the results of these evaluations confirm the ones in Figure 3.20. Also in this case, increasing the size of the critical sections led to increased blocking times and thus to increased response time values, the impact of non-preemptive scheduling and blocking being significant for all tasks in the system. The over-proportional growth of the WCRTs in case of bursty task activations is in this case even more obvious for the lowest priority tasks in the system, i.e. τ_5 and τ_6. These are

strongly affected by the bursty activations of the higher priority local tasks. Depending on the tasks' deadlines such an increase may eventually lead to deadline misses.

3.11.2.2 Invidual Contribution of different Factors to the WCRTs and Impact of Non-Preemptive Blocking on the Processor Core Utilization

In order to highlight the individual contribution of different factors to the tasks worst-case response times we provide in Figure 3.22a) the results for one particular test case configuration (see Table 3.4) and the critical sections setups for 5%, 15%, and 25% of the tasks WCETs.

Table 3.4: Particular configuration of the parameters for the system in Figure 3.1b under partitioned SPNP scheduling and MLP-NP shared resource arbitration.

Mapping	Task Name	Priority	Period T_i (ms)	Core Execution Time C_i (ms)	Global Resource Accesses $n_i^G * \omega_i^G$	CS $25\% \cdot C_i$
Core 1	τ_1	1	300	75	2 * 9.375 to GR1	18.75
Core 1	τ_3	3	188	47	2 * 5.875 to GR2	11.75
Core 2	τ_4	4	280	70	2 * 8.75 to GR1	17.5
Core 2	τ_6	6	200	50	2 * 6.25 to GR2	12.5
Core 3	τ_2	2	108	27	2 * 3.375 to GR2	6.75
Core 3	τ_5	5	440	110	1 * 13.75 to GR1 1 * 13.75 to GR2	27.5

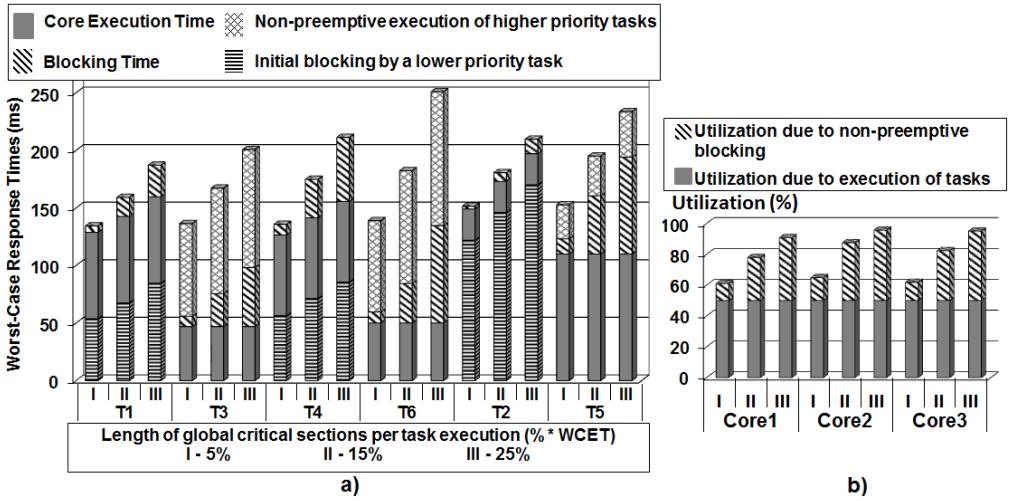


Figure 3.22: a) Worst-case response time of the individual tasks and b) utilization of the individual cores depending on the critical sections length.

Worth to mention here is the influence of the blocking times on the cores' utilization levels. In case of non-preemptive scheduling the blocking times behave like an extension of the tasks' execution times and thus contribute to the core utilization. But the blocking times of a task actually depend on the execution of other tasks mapped on other cores using the same shared resources. This makes the core utilization a function of the critical

section lengths of the other tasks on the other cores. As depicted in Figure 3.22b) the "effective" utilization level of the cores increases with the length of the critical sections being at least 10% higher than the core default utilization (i.e. 50%) when the length of the critical sections is 5% of the tasks WCETs, and rising up to 96% in case of Core 2 when the critical sections are 25% of the tasks WCETs.

In single-core non-preemptive setups the influence of sharing resources could be neglected due to the intrinsic behavior of the non-preemptive scheduler which avoids the synchronization overhead due to resource sharing mechanisms (see Section 3.5.1). But, when implementing non-preemptive scheduling on multi-core systems the effect of sharing resources on the response times of the tasks and on the utilization levels of the processor cores can not be neglected anymore. While in single-core implementations non-preemptive scheduling policies are advantageous by reducing context switching cost where response times are not critical, multi-core setups show an increased load which threatens schedulability beyond deadline violations. This effect constrains the migration of non-preemptive task sets from single-core to multi-core systems. As demonstrated in this evaluation section, the analysis solution presented in Section 3.8 allows the derivation of task blocking and response times which can be used to guide the decisions of the designers regarding the implementation and the mapping of real-time applications on non-preemptively scheduled multi-core systems.

3.11.3 Evaluation of AUTOSAR conform Multi-Core Setups

In this section we present the evaluation of the analysis approach introduced in Section 3.9 and show its applicability to AUTOSAR conform automotive multi-core systems. For evaluation we consider the analysis of the timing behavior of a set of pseudo-generated test cases for two multi-core setups namely, the dual-core system in Figure 3.15 and the multi-core system in Figure 3.23.

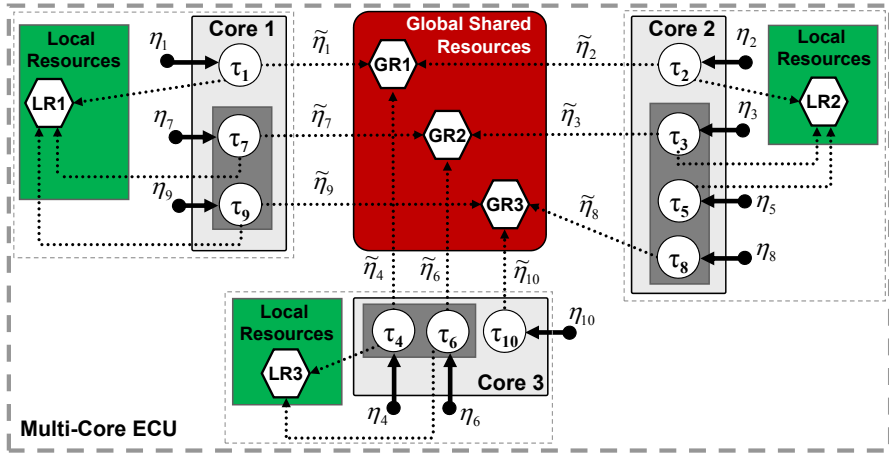


Figure 3.23: Multi-core ECU with tasks accessing local and global shared resources.

Our goal is to investigate the timing behavior (i.e. response-times) of these two setups under AUTOSAR spinlock-based synchronization mechanism in combination with different AUTOSAR OS conform core local scheduling policies, namely: (i) fully preemptive (FP), (ii) cooperative (Coop), iii) mixed-preemptive (MP) and (iv) fully non-preemptive (FNP). Fully preemptive means that on each core there is no group of tasks and the scheduling policy is static-priority preemptive. Cooperative corresponds to the setups depicted in Figure 3.15 and Figure 3.23 where tasks in each group are scheduled cooperatively to each other (i.e. are preemptable at runnables border among each other and everywhere preemptable for tasks with higher priorities which are not in the same group). Mixed-preemptive corresponds to the setups depicted in Figure 3.15 and Figure 3.23 where tasks in each group are scheduled non-preemptively to each other and preemptively to the other tasks. Finally, fully non-preemptive means that on each core there is one group containing all tasks and the scheduling policy is static-priority non-preemptive.

Basic Configuration Parameters.

Beside the number of cores, the difference between the two setups is given by the mapping of tasks and therewith by the load on the individual cores. Thus, in both setups we considered the same tasks with the same priorities, same number of equally long runnables, accessed local and global shared resources and OSEK-group membership. Table 3.5 summarizes the key configuration parameters of the tasks in the two system setups.

Table 3.5: Particular configuration for the systems in Figure 3.15 and Figure 3.23

Task[1,2]	OSEK Group	GR Accesses n_i^G / runnable	LR Accesses n_i^L / runnable	Mapping[3]
τ_1	-	2 to GR1	2 to LR1	Core1
τ_2	-	2 to GR1	2 to LR2	Core2
τ_3	with τ_5, τ_8	2 to GR2	2 to LR2	Core2
τ_4	with τ_6	2 to GR1	2 to LR3	Core1/Core3
τ_5	with τ_3, τ_8	-	2 to LR2	Core2
τ_6	with τ_4	2 to GR2	2 to LR3	Core1/Core3
τ_7	with τ_9	2 to GR2	2 to LR1	Core1
τ_8	with τ_3, τ_5	2 to GR3	-	Core2
τ_9	with τ_7	2 to GR3	2 to LR1	Core1
τ_{10}	-	2 to GR3	-	Core2/Core3

[1] - Priority indicated by task index, lower index means higher priority.
[2] - Each task comprises three equally long runnables.
[3] - Load per Core is 75% in DC setup and 50% in MC setup.

Similar to Section 3.11.1.2 and 3.11.2.1, the activation period, the activation jitter, and the worst-case execution time (WCET) per task were generated according to the UUnifast algorithm [19] as follows: the utilization on each core was assumed to be $U_{core} = 75\%$

in the dual-core setup in Figure 3.15 and $U_{core} = 50\%$ in the multi-core setup in Figure 3.23; based on the assumed core utilization each task on a core was assigned a random utilization U_i such that the sum of all task utilizations on that core equals the total core utilization ($\forall \tau_i$ mapped on $core : \sum U_i = U_{core}$); tasks' activation periods P_i were generated randomly between 100ms and 1000ms; each task was randomly assigned an input jitter from the interval $[0, 2 \cdot P_i]$ (i.e. each task could potentially be activated by a maximum burst of 3 simultaneous activations); based on the chosen periods the tasks WCETs C_i were assigned to match the task utilization U_i. For each task, its WCET was equally split among the three comprised runnables. The size of critical sections were generated as will be described below for the individual experimental evaluations.

Experiment 1. For the first evaluation we randomly generated system configuration for the dual-core (DC) and multi-core (MC) setups as follows. In a first step the activation period, the activation jitter, and the worst-case execution time of each task and each runnable was generated as described above. In the second step, the size of each critical section was generated as follows: the total size of critical sections per task CS_{total} was generated randomly to be a percentage value $x\%$ ($x \in \mathbb{N}$) between 1% and 90% of the shortest WCETs among all tasks in the system, i.e. $CS_{total} = x\% \cdot \min(C_i), \forall \tau_i$; then, the total size of critical sections was equally split among the maximum number of critical sections per task instance, in our system setup in Table 3.5 this being 12 (three runnables with maximum four critical sections per runnable). Thus, the size of each critical section of each task τ_i was randomly obtained with $\omega_i^{GR} = \omega_i^{LR} = x\% \cdot \min(C_j)/(3 \cdot max(n_k^G + n_k^L))$, where the different indices i, j, k indicate the fact that for a certain system configuration the size of the critical sections potentially depends on the parameters of other tasks in the system [26]. In this way we ensure that the sum of the critical sections per runnable never exceeds the size C_{r_i} of the runnable.

The minimum distance between requests d_{srr} in (3.9) was applied at runnable level, i.e. for each runnable d_{srr} was set up to be $d_{srr} = (C_{r_i} - \omega_i^{GR})/(n_i^G + n_i^L - 1)$, i.e. accesses for shared resources are equally spread across C_{r_i}.

With this procedure we generated system parameters until we got 5000 schedulable system configurations in the dual-core (DC) and multi-core (MC) setups, each under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) AUTOSAR OS. The worst-case response times (WCRTs) of the tasks, parametrized randomly as discussed above, are illustrated in Figure 3.24a) to f). The WCRTs are given as mean values over the 5000 configurations.

The first aspect that can be observed when looking at the obtained analysis results in Figure 3.24a) to f) is that the distribution of the load across multiple cores generally leads to lower task WCRTs, i.e. task WCRTs in the multi-core setup are in general lower when compared to the WCRTs in the dual-core setup. The only exception can be observed in case of the highest priority task τ_1 on Core 1 which, in the multi-core

[26] Example for τ_5: Assume $C_5 = 11$, $x = 10\%$, $min(C_j) = C_8 = 10ms$ and $3 \cdot \max(n_k^G + n_k^L) = 3 \cdot (n_1^G + n_1^L) = 3 \cdot 4 = 12$. Thus, $\omega_i^{GR} = \omega_i^{LR} = 0.083ms$, $\forall \tau_i$ and therewith $\omega_5^{GR} = \omega_5^{LR} = 0.083ms$.

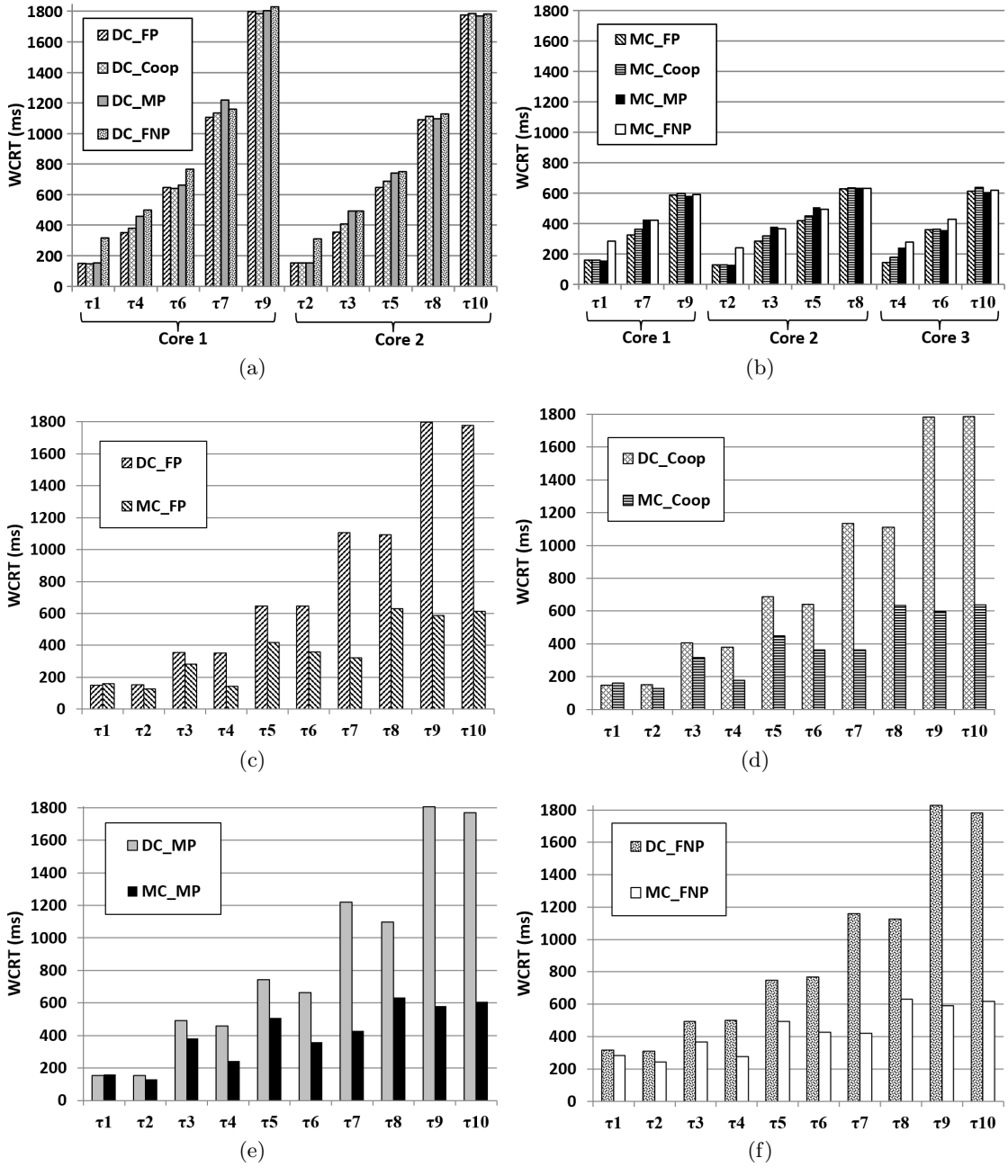


Figure 3.24: WCRTs under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) scheduling for randomly generated parameter for the dual-core (DC) and multi-core (MC) setups in Fig 3.15 and Figure 3.23.

setup under FP, Coop and MP scheduling, experiences slightly increased WCRTs (see Figure 3.24c), d) and e)). This is not a surprising result. In the multi-core setup, task τ_1 is additionally directly blocked by the remote task τ_4, in comparison to the dual-core setup where τ_4 was a lower priority local task. As τ_1 has the highest priority in the system and is not part of any group under FP, Coop and MP scheduling it is less influenced by the core local scheduling policy, but more by the blocking effects. However, under FNP scheduling, the core local scheduling effects dominate the blocking effects and thus the WCRT of τ_1 in the multi-core setup is lower than in the dual-core setup

These aspects, observed for task τ_1, do not occur for task τ_2, the highest priority task on Core 2. For task τ_2, independent of setup, task τ_4 is always a directly blocking remote task and thus, τ_2 generally takes advantage of the reduced core load in the multi-core setup. For the rest of the tasks, independent on scheduling policy the multi-core setup enables reduced WCRTs, the improvement going up to 65% for task τ_{10} and 67% for task τ_9 (see e.g. Figure 3.24f).

When comparing the scheduling policies in the dual-core and multi-core setups (see Figure 3.24a and b), one can see that FNP is never the best system-wide option. Under FP, Coop and MP scheduling, the WCRTs of the tasks with the highest priorities in the system, i.e. τ_1 and τ_2, are just slightly varying. Being always able to preempt the other tasks on their cores, the small differences between the FP, Coop and MP setups are essentially given by the blocking times τ_1 and τ_2 experience under different combinations of core local scheduling and regions with disabled interrupts (see Section 3.9.2.1) under AUTOSAR spinlock-based arbitration.

For the tasks τ_3, τ_4, τ_5, τ_6, τ_7, τ_8 and τ_9, which under Coop and MP scheduling are clustered in groups and scheduled either cooperatively or non-preemptively to each other, the impact of the core local scheduling policy can be clearly observed in Figure 3.24a and b). For example, in both setups the WCRT of task τ_4 under Coop and MP scheduling increases in comparison to FP scheduling because of the non-preemptive regions of the lower priority task τ_6. Under Coop scheduling τ_6 is preemptable to τ_4 only at runnables borders and under MP scheduling the execution of τ_6 is completely non-preemptive to τ_4. Therefore, whereas the WCRTs of τ_4 obviously increase under Coop and MP scheduling, task τ_6 takes advantage of the non-preemptive execution against τ_4 in the sense that its WCRT is lower or only slightly increases in comparison to the FP scheduling. The same behavior can be observed for the group comprising the tasks τ_7 and τ_9 on Core 1 and for the group comprising the tasks τ_3, τ_5 and τ_8 on Core 2. Whereas the WCRTs of the tasks τ_3, τ_5 and τ_7 are clearly larger under Coop and MP scheduling in comparison to FP scheduling, the WCRTs of the lower priority tasks τ_8 and τ_9 do not significantly change.

Finally, for task τ_{10}, the lowest priority task in the system, the WCRTs in the four investigated setups are just slightly varying, the differences being actually given by the system-wide blocking scenarios and not by the core local scheduling policies.

From all these results, one can also see that on each core the WCRTs of the tasks are raising while the priorities decrease, behavior that corresponds to the expectations of the automotive priority based design.

Experiment 2. In a next experiment we generated configurations similar to experiment 1, with the difference that for each test case we varied the length of the critical sections from 1% to 25% of the tasks' worst-case execution times (WCET). This means that we didn't search for the shortest WCET but for each task τ_i we individually generated the size of critical sections depending on its randomly generated WCET C_i. We applied the response-time analysis methods to multiple test cases until we got 1000 schedulable configurations [27] for the dual-core (DC) and multi-core (MC) setups, each under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) scheduling.

The eight diagrams in Figure 3.25 depict the worst-case response times (WCRTs) of the tasks depending on the critical sections' length for the DC and MC setups under the four scheduling options. The WCRTs are given as mean values over 1000 configurations.

As expected, increasing the size of the critical sections led to increased blocking times and thus to increased response time values. From the perspective of each task, the increased critical sections length causes an increased delay not only on its own execution but also on the execution of the lower and of the higher priority local tasks. These delays are also parts of the tasks' WCRTs.

As can be seen in the diagrams of Figure 3.25 the results of the second evaluation confirm the results of the first one. This means, the tasks with the highest priorities, τ_1 and τ_2, are in general less influenced by the scheduling strategy and by the number of cores, their WCRTs slightly increasing with the size of critical sections. The tasks with intermediate priorities, i.e. τ_3, τ_4, τ_5, τ_6 and τ_7, which under Coop and MP scheduling are clustered in scheduling groups, experience lower WCRTs in the MC setups when compared to the DC setups, their WCRTs linearly increasing with the size of critical sections in all setups and under all scheduling policies. The tasks with the lowest priorities on all cores, i.e. τ_9 and τ_{10} in the DC setups and τ_8, τ_9 and τ_{10} in the MC setups, are the most impacted tasks, their WCRTs significantly raising with the increase of the critical sections size. Depending on the tasks' deadlines such an increase may eventually lead to deadline misses. However, similar to the other tasks, the lower priority tasks experience lower WCRTs in the MC setups in comparison to the DC setups. Once again, FNP scheduling is never a good system-wide option.

The lower WCRTs obtained in the multi-core setups, in the first and the second experiment, confirm the expected and the desired benefit of distributing the load across multiple cores.

The tests (non-optimized code) for the first and the second experiments were performed on an Intel Core i7-3517U 1.90 GHz CPU, 10GB RAM, 64bit Windows and took in average 125ms for one analyzed configuration.

[27] 2 setups x 4 scheduling options x 25 CS setups x 1000 schedulable configurations means 200000 successfully analyzed system setups.

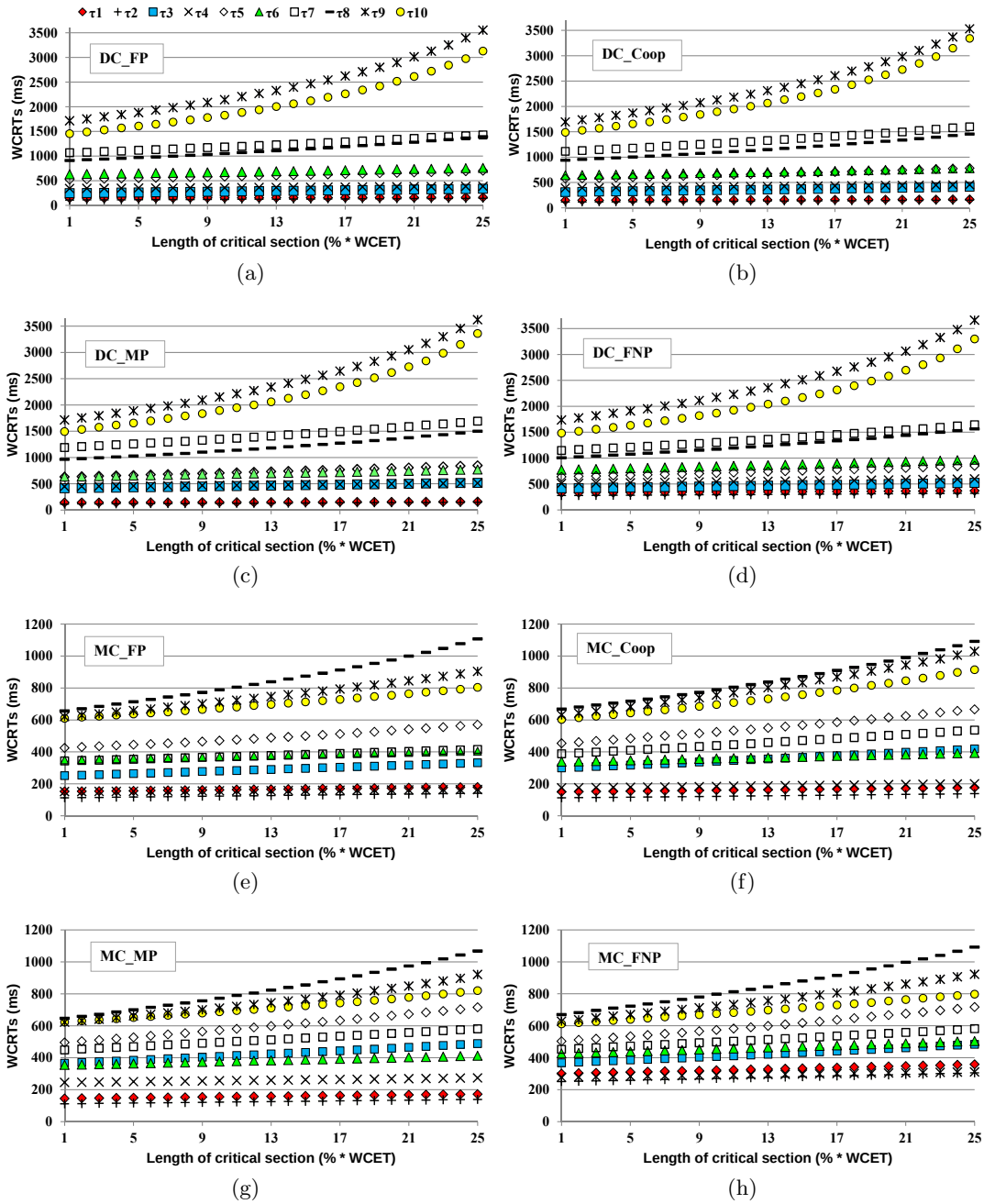


Figure 3.25: WCRTs depending on the critical sections length in the dual-core (DC) and multi-core (MC) setups under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) AUTOSAR scheduling. (*Note the difference between the scale range in case of DC and MC analysis results*).

3.12 Summary

This chapter addressed the timing behavior of multi-core systems with shared resources. First, key components of a safe synchronization algorithm for shared resources in multi-core systems were discussed and the interdependence between design decisions regarding task scheduling and shared resource arbitration was emphasized. These highlight the fact that only a complete specification of the arbitration and scheduling policies enable a reliable and predictable timing behavior of multi-core systems.

Further, novel worst-case blocking-time and response-time analysis methods for real-time applications mapped on partitioned multi-core setups with shared resources were proposed. These methods support different processor scheduling policies and shared resource arbitration strategies, proposed by academia and industry, consider realistic applications models with tasks that exhibit arbitrary activations and deadlines, and rely on an enhanced model to capture the load imposed on shared units. More exactly, a timing analysis solution was proposed first for partitioned multi-core setups where processor cores are individually scheduled according to the static-priority preemptive (SPP) policy and shared resources are arbitrated according to the Multiprocessor Priority Ceiling Protocol (MPCP) [116]. In a next step, partitioned static priority non-preemptive (SPNP) scheduling was addressed in the context of multi-core setups in combination with a spinlock-based synchronization mechanisms and a corresponding timing analysis approach was introduced. These two steps paved the way for a timing analysis method that apply to automotive AUTOSAR conform multi-core processors. For such setups, the combination of partitioned fixed-priority AUTOSAR OS scheduling, which specify preemptive, non-preemptive and cooperative core local scheduling, and lock-based shared resource synchronization using the Priority Ceiling Protocol [100] for core local shared resources and a spinlock-based arbitration mechanism for inter-core shared resources [12] was handled.

In order to tackle the contention of tasks on the processor cores and on the shared resources, the blocking-time and response-time analysis equations were integrated in the iterative analysis procedure of the compositional system-level performance analysis methodology discussed in Chapter 2. Section 3.10 showed that all analysis elements comply with the conditions of the fixed-point theory regarding the convergence of iterative analysis procedures, fact that enables the calculation of conservative (i.e. safe) analysis results. The experimental part demonstrates the applicability and usefulness of the proposed analysis solutions.

Note that, following the principle “Cooperate on standards, compete on implementation” AUTOSAR mandates the availability of spinlocks for inter-core synchronization, but doesn’t specify implementation details on the execution of conflicting critical sections. The order of granting the locks is one essential design decision for obtaining predictable timing behavior. For the purpose of this thesis spinlocks were assumed assigned based on tasks priorities. This decision was taken to maintain the compatibility with the state-of-the art priority based scheduling in the automotive design, however, the proposed analysis framework is conceived to be adapted to other design decisions.

4 Timing Analysis of Multi-Mode Applications on Multi-Core Systems

4.1 Introduction

Acting in an complex environment that consists of diverse physical elements (e.g. natural environment, infrastructure, transportation, telecommunication, energy systems) and often of humans participants, many real-time embedded systems are required to change their functionality over time and execute in different operational modes. Safety-critical avionic and automotive control systems or multimedia smart devices, are just few examples of real-time systems that may have to adapt their behavior during runtime to changing conditions in the environment, to switch to an emergency state or to change their resource usage. Such systems are called multi-mode systems and the applications running on them are called multi-mode applications [118].

Besides the implicit need for an adaptive behavior of some real-time systems, another important reason for implementing different operational modes is to save costs by integrating an increasing amount of applications on a reduced number of computational resources. In this case multiple operational modes have to be defined and configured to exclusively make use of the available resources in order to limit the maximum load on the systems. An example, also from the automotive domain, are the driving mode features (e.g. Economy Mode, Comfort Mode, Sport Mode, Offroad Mode) that are implemented in current generation of passenger cars for example from BMW, Daimler or VW [1]. Changes between such operational modes are not the result of environmental changes, but the result of explicit commands of the drivers. As switching between such operational modes are occurring not only when the cars are standing but also while driving, and because such switches imply simultaneous changes of multiple car parameters (e.g. steering, gearbox, accelerator and brake pedal, and engine parameters) the transition between modes has to be realized in a safe and fast manner.

To properly handle mode transitions, operational modes and mode change protocols which control the transition between the modes have to be defined. Each mode is characterized by a different behavior and is associated with a specific set of tasks together with its timing properties, e.g. task execution times, priorities and deadlines. During the transition between modes, some tasks can be stopped or simply aborted, new tasks can be activated or, in case there are multiple processing resources (i.e. processors), some tasks can be migrated. Additionally, there can be tasks which are present in multiple modes

[1] When using the online car configurator on the car manufacturers website, such features can be identified as part of the default configurations or as an extra feature that can be selected by customers.

and execute independent of the transition. The combined execution of all these types of tasks might lead to an increased workload on the processors and therewith to potential deadline misses of some task. Therefore, when a multi-mode system is part of a hard real-time embedded system it is imperative to guarantee that timing constraints are not violated at any moment of the systems execution, i.e. neither in the individual modes, nor during the transition phases between modes [118, 65, 145, 89]. Consequently, it is essential to provide designers of multi-mode real-time systems with appropriate methods for the verification of timing constraints. In addition to the verification procedures, new design solutions are required in order to enable fast (e.g. within few milliseconds) and at the same time controlled and safe transitions between modes.

Furthermore, as highlighted in the previous chapters, multi-core architectures emerge as the preferred platform for embedded real-time applications. For the automotive domain, this trend is confirmed by the increasing offer of multi-core processor solutions [66, 50] and by the AUTOSAR standard which introduced support for partitioned multi-core OS [12]. In the meantime the AUTOSAR standard also introduced guidelines for mode management [10, 13]. Both can co-exist, so the problem of designing and analyzing multi-mode systems has to be handled in the context of multi-core systems. Appropriate mechanisms that jointly handle the (i) mode management, (ii) multi-core scheduling and (iii) shared resource arbitration are required in order to ensure correct system functionality. Consequently, proper timing analysis methods are needed for the prediction of the timing behavior of multi-mode applications on multi-core systems.

Related work addressing one of the three topics, i.e. mode management, multi-core scheduling and shared resource arbitration, is already available.

Several mode change protocols and dedicated timing analysis methods have been developed for handling mode transitions in multi-mode single-processor [139, 153, 103, 118, 145] and multi-processor systems [95, 161]. Most of the existing solutions consider only applications without communicating tasks, i.e. assume only independent tasks and neglect communication precedence relations between them. However, many real-time systems are composed of multiple processors and accommodate distributed applications which consist of multiple communicating tasks. The research presented in [65] showed that in such systems, the initiation of a mode change has not only a local effect on one processor but also impacts the timing of tasks executing on other processors. Transient overload situations, caused by a mode change, can recurrently propagate as “waves” between the components of a system (i.e. busses and processors) and thus challenge the real-time behavior of the entire system. The common assumption of all existing approaches, including the one developed in [65], is that a transition between two modes can be initiated only when the system is running in a steady state corresponding to one operational mode, i.e. the overlap of multiple mode changes is not allowed. However, the duration of the transition phase, called “settling time of a mode change” or “mode change transition latency”, has to be bounded in order to guarantee that a system has reached a steady state after a mode change. Solving this problem is key in order to enhance the predictability of real-time systems’ behavior, e.g. in the automotive and avionics domain. Providing an analysis approach that can be used for computing upper

bounds on the mode change transition latencies for distributed applications is therefore one of the main contributions of this chapter.

With respect to timing analysis solutions for multi-core scheduling and shared resource arbitration, Chapter 3 highlighted a large amount of related work on these topics. However, despite the practical relevance for real-life applications, none of the existing solutions addresses the complex setup consisting of multi-mode applications that share resources on multi-core systems. Therefore, the second main contribution of this chapter consists of approaches for safely handling shared resources across mode changes in multi-core setups and of corresponding timing analysis methods.

In what follows, Section 4.2 discusses related work on multi-mode systems. Section 4.3 introduces a general system and mode change model. Section 4.4 discusses challenges in predicting the timing behavior of multi-mode distributed systems and introduces a solution for bounding mode change settling times (i.e. mode change transition latencies). Further, Section 4.5 presents approaches for handling shared resources across mode changes in multi-core systems with shared resources and introduces corresponding blocking-time and response-time analysis methods. Experiments introduced at the end of Section 4.4 and 4.5 demonstrate the applicability of the proposed approaches.

4.2 Related Work

The problem of scheduling multi-mode systems and of analyzing their timing behavior across mode changes has been addressed previously. A comprehensive survey of mode change protocols and associated analysis methods is provided in [118].

In literature, the tasks executing in a multi-mode system are categorized depending on their behavior when a Mode Change Request (MCR) occurs. Thus, there are: (i) *old-mode tasks*, which are immediately aborted when a MCR occurs (ii) *old-mode finished or completed tasks*, which are present in the old execution mode, but not in the new one. In order to ensure data consistency or correctness of future executions these tasks are allowed to finish their execution during the transition phase which follows the MCR; (iii) *new-mode or added tasks*, which are either introduced for the first time after the MCR or represent a modified version of old tasks, e.g tasks that change their parameters - execution time or activating event model; (iv) *unchanged tasks* which are present in both configurations and remain unchanged in their parameters in each operational mode and during the transitions between them.

With respect to the way in which periodically activated unchanged tasks are executed during transitions, two types of protocols were defined, namely: *with periodicity*, where their activation pattern is preserved independent of the mode change in progress and *without periodicity* where the perioridicity may be altered during the transitions.

Depending on whether new-mode and old-mode tasks may coexist during the transition phase which follows a MCR, mode change protocols are categorized in *synchronous protocols* and *asynchronous protocols*. *Synchronous protocols* do not allow new mode tasks to be released until all finished tasks have completed their last activation corresponding to the old mode. In this way, synchronous protocols ensure isolation between

the execution of mode specific functionalities. These protocols are generally simple and do not require specific schedulability analysis for the transition phase. However, they are not very prompt and delaying the start of the new mode tasks is not always suitable, e.g. when switching to an emergency mode where new mode tasks must be performed as soon as possible. Alternatively, *asynchronous protocols* overcome this limitation by allowing new-mode tasks execute in parallel to the old-mode tasks. However, the overlap of old and new mode tasks generates an increased workload during the transition phase and can potentially lead to timing violations [104, 118, 65]. Therefore, asynchronous protocols require specific schedulability analysis.

Note that the AUTOSAR specifications related to the mode management topic indicate the support for synchronous as well as asynchronous mode change protocols [10, 13].

Corresponding to the different mode change protocols, several timing analysis solutions were proposed starting 1989. In [139] an analysis approach is proposed for mode changes on single-processor systems scheduled according to the rate-monotonic scheduling policy. In [153], the authors show that the analysis in [139] is not sufficient, because the test may pass a task set that is unschedulable. They improve the previous analysis and propose a new one which considers deadline monotonic scheduling.

A new model for mode changes which avoids overload situations by considering offsets when performing mode transitions is introduced in [104] and [103]. However, an algorithm for offset calculation is not provided. Another mode change protocol and an algorithm for computing the offsets required to delay the initiation of mode transitions in order to avoid overload situations is introduced in [118]. All these solutions are limited to strictly periodic task activations. An analysis method for multi-mode single-processor systems which consider fixed-priority and EDF scheduling and arbitrary task activation patterns is presented in [145]. For systems that are initially proven not schedulable during the transition phase, the approach in [145] derives offsets for delaying the start of transition between two modes in order to make the system schedulable.

In [108] the authors introduce a framework for the compositional analysis of real-time systems which execute multiple-mode applications concurrently under a hierarchical scheduling policy on a single processing resource. A semantic framework for the specification and analysis of mode change protocols was presented in [107].

Mode changes in the context of hierarchical component-based design was addressed in [159, 160]. [159] proposed a mode switch logic that ensures that multiple components of a multi-mode system can perform a mode change in a synchronized and coordinated manner such that the entire system is in a consistent state after switching modes. This logic assumes that the execution of each component is immediately aborted when a mode switch is triggered. This logic was extended in [160] to support for atomic component execution, i.e. for systems where atomic components and atomic execution groups (comprising multiple components) cannot be interrupted by a mode switch and have to complete any ongoing execution before reconfiguration for the new mode. As a solution to avoid conflicts between multiple mode change requests, the mode change logic can discard a new MCR or delay it until the completion of an ongoing mode change. In

order to do that the mode change timing is required. A very basic mode change timing analysis was also proposed in [160]. This assumes the timing of each mode change step represented by a known constant value (e.g. the transition time of each component) and neglect issues related to scheduling and data transmission.

All related work mentioned above assumes only independent tasks or neglects communication precedence relations between them. However, many real-time systems provide complex functionalities by accommodating distributed applications mapped across multiple processing units.

An approach for handling mode changes in the context of pre-runtime scheduling for time-triggered distributed hard real-time system was presented in [48]. The approach essentially supports mode changes at runtime by switching through a series of off-line calculated transition schedules, that prepare for the new mode. In case of a mode change requests, it is checked whether discontinuing the currently running schedule (old-mode schedule) and immediately starting the transition is feasible, i.e. if stopping activities of current (old) mode frees enough CPU-time for the activities related to the transition schedule. If this is not the case, the mode change is performed after the old-mode schedule has completed execution. The maximal delay is considered in the offline construction of the schedules [48].

Further solutions addressing the problem of mode changes in distributed real-time system were presented in [103, 65, 144, 89]. The analysis approaches presented in [103] do not consider communication precedence relations between tasks when reasoning about the timing of the tasks during transition phases. Timing implications of mode changes in distributed real-time systems with communicating tasks were discussed in [65, 144] and [89]. Firstly, [65] proposed a method for computing the WCRTs of tasks during the transition phases between two modes of a distributed system. Further [65] showed that in case of distributed applications, the initiation of a mode change has not only a local effect but also impacts the timing of tasks executing on other processors. The mode change leads to a change in the execution and communication demand on a processor. As there are tasks which communicate across the processors, the transient timing behavior of tasks on a processor during the transition phase will propagate to the interconnected tasks and will impact the timing of other processors. This transient effect, initiated on a processor may occur on other processors long time after the mode change was performed. The main issue is that most existing solutions, including the one in [65], assume that a mode change requests can be served only when a system executes in a steady state corresponding to one operational mode, however, without indicating when a system executes in a steady state. Therefore, computing only the WCRTs in each individual mode and during every transition between two modes is not enough. The duration of the transition phases has to be computed and considered at design time. The latency of a mode change for single-processor and distributed systems was addressed in [103], however, without considering the recurrent effect of a mode change that occurs in setups where tasks communicate across cores. [89] proposed the first analysis algorithm which gives a maximum bound on each mode change transition latency of multi-mode distributed applications thereby overcoming limitations of previous work. The contribution of [89]

is subject of Section 4.4.

Mode change protocols and analysis solutions were proposed also for multiprocessor platforms under global and partitioned multiprocessor scheduling. In [95] a synchronous and an asynchronous mode-transition protocol without periodicity (i.e. does not consider unchanged/mode-independent tasks) were introduced for identical multiprocessor systems under global preemptive and fixed job-level priority scheduling. These protocols were extended to uniform multiprocessor platforms in [162]. A mode-change protocol and a corresponding analysis for multi-mode multiprocessor systems with periodicity (i.e. considers mode-independent tasks) under global EDF scheduling was presented in [94]. The problem of changing modes under multiprocessor partitioned EDF scheduling on identical multiprocessor platforms was addressed in [57]. For such setups, two methods were proposed for handling mode changes in the context of a synchronous mode change protocol with periodicity. The first method consists in computing an offline static allocation of tasks on processors such that synchronous mode changes can be safely performed. The second method proposes sufficient conditions for verifying whether online task allocation leads to feasible schedules and satisfies task transition deadlines [57].

As in the case of single processor systems, related work on multi-mode multiprocessor systems mainly assumes applications consisting of independent tasks, i.e. tasks which don't communicate or without precedence constraints between them. However, providing support for handling shared resources in multi-mode setups is essential for the design process of real-life embedded real-time applications, as for example for the next generation AUTOSAR conform automotive multi-mode multi-core applications [2].

The problem of sharing resources by multi-mode applications was studied in [139, 153, 118] but only for single-processor systems. For *asynchronous mode change protocols* [118], where *new mode tasks* may interfere with *old mode tasks*, it was shown that the classic Priority Ceiling Protocol (PCP) (in its original form or in the form of the Immediate PCP) which is based on static task priorities and on a procedure of dynamically adjusting shared resource priority ceilings, is not directly applicable [139, 153, 118] and counter the safe system functionality.

The main issues concern the procedure of adjustment of the shared resources ceilings *across asynchronous mode changes* [118] [3]. When switching from an *old operational mode* to a *new operational mode* as a consequence of a mode change request (MCR), *asynchronous mode change protocols* allow *new mode tasks* to be released before all *old mode tasks* have completed their last activation corresponding to the old mode. Depending on the tasks priorities and therewith on the share resource ceilings two problems can occur: (1) if ceilings have to be raised but are adjusted too late then a new mode task may inherit an old mode ceiling which is lower than the current task priority. This violates the ceiling based protocols, as ceilings must never be lower than the priority of any

[2] The AUTOSAR standard introduced independent guidelines for mode management [10, 13] or sharing resources in multi-core setups [11].

[3] In case of synchronous mode change protocols, sharing resources does not introduce problems since old-mode (finished) tasks and new-mode (added) tasks are executed separately and ceiling priorities can be adjusted after finishing the old-mode tasks and starting the new-mode tasks [118].

task using the resource; (2) if ceilings have to be lowered but are adjusted to early then an old mode task may inherit the new mode lower priority ceiling. Thus, activations of the old mode tasks, executed after the MCR, could experience increased blocking in comparison to the activations executed before the MCR. Both situations invalidate the blocking time analysis.

[139] proposed a set of strict rules to determine when new-mode tasks can be added to the system and when ceiling priorities can be adjusted. The rules for raising and lowering priority ceilings ensure that a task cannot be blocked more than once by a lower priority task. However, these rules combined with the scheduling rules for starting new-mode tasks reduce the overall system responsiveness to the mode change requests. Furthermore, [153] showed that the protocol and the corresponding analysis in [139] are insufficient and may allow unfeasible systems pass the schedulability analysis.

A general solution proposed in literature for avoiding the problem caused by the need to dynamically adjusting ceiling priorities in single-processor systems is to define for each semaphore which protects a shared resource one priority ceiling, called *"ceiling of ceilings"*, which is valid for all operating modes [139, 153, 118]. Using this protocol, any semaphore receives a priority ceiling equal to the highest priority of any task accessing it before or after the mode change. As these priority ceilings remain unchanged through the applications lifetime, the problem of dynamically adjusting them during mode changes is avoided. The disadvantage of this solution is that it can easily lead to excessive blocking times. By simultaneously considering all possible operational modes of a systems, the ceiling priorities will be often set too high [139, 153, 118] for individual mode and thus generate unnecessary blocking scenarios.

Although the problems and the mentioned solution for handling shared resources in multi-mode setups are stated in the context of single-processor systems, they are also valid for multi-mode multi-core systems. However, the complex setup consisting of multi-mode applications that share resources when executing on multi-core systems was neglected until recently. The need for appropriate mechanisms that jointly handle the (i) mode management, (ii) multi-core scheduling and (iii) shared resource arbitration was identified in [91] and [92]. To fill the existing gap, [92] proposed an approach for safely handling inter-core and intra-core shared resources across asynchronous mode changes and a corresponding blocking- and response-time analysis approach. The contribution of [91] and [92] is subject of Section 4.5.

4.3 System and Mode Change Model

Relying on the general system model in Section 2.2 this section introduces model elements of a multi-mode system which provide basis for the timing analysis solutions described in the following sections of this chapter.

According to the general system model in Section 2.2 a real-time system is assumed composed of a set of computation and communication tasks $\mathcal{T} = \{\tau_1, \dots \tau_n\}$ which are statically mapped and executed according to an arbitration strategy on a set of processing (CPUs) and communication (Buses) platform elements (resources).

Such a system may execute in different operational modes specified by a finite set $M = \{M_1, M_2, \dots M_z\}$ ($z \in \mathbb{N}$). Each mode $M_i \in M$ is characterized by a different behavior and is associated with a specific set of tasks (a subset of $\mathcal{T}$) that are active in that mode. The possible transitions between two modes in M are specified by a finite set $\Phi = \{\Phi_{M1}^{M2}, \dots, \Phi_{M_y}^{M_z}\}$. A transition between two modes is initiated by a mode change request (MCR) triggered by the environment or by system internal requirements. The MCR is assumed as an global atomic event which may be triggered at any moment t_{MCR} during runtime. In order to exclude interference of multiple mode changes, a new MCR can be served only if the system is not executing a transition between two modes as a result of a previous MCR. The execution of a mode change as a consequence of the MCR is controlled by a mode change protocol [4]. In this thesis we focus on *asynchronous mode change protocols* [118, 13] and consider during transitions the following types of tasks:

- *finished tasks* (denoted τ_{iF}) whose jobs/instances activated before t_{MCR} are allowed to finish after the occurrence of the MCR;
- *added tasks* (denoted τ_{iA}), which are activated with an offset $\phi_{\tau_{iA}}$ after the MCR (i.e. at $t_{MCR} + \phi_{\tau_{iA}}$) and thus execute only in the new mode. The offset $\phi_{\tau_{iA}}$ is assumed to be a constant value known for each added task τ_{iA};
- *unchanged tasks* (denoted τ_{iU}), which execute in both modes without any change in parameters.

The first index associated to the task τ_{iF}, τ_{iA} or τ_{iU} stands for priority and the second indicate its type, i.e. finished, added and unchanged, respectively.

For illustrative purpose consider the example in Figure 4.1, which depicts a multi-mode distributed system in a steady mode M_1, during the transition phase from the (old) mode M_1 to a (new) mode M_2 and finally in the steady mode M_2.

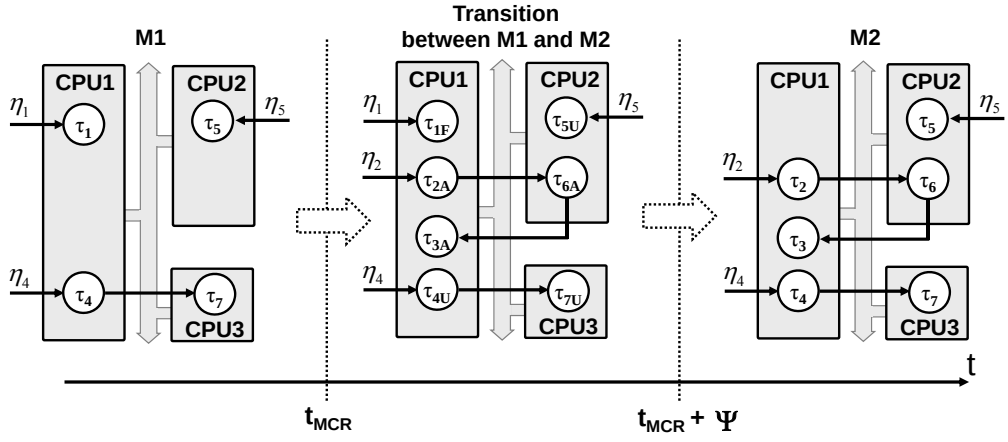


Figure 4.1: Distributed system performing a transition between two modes M1 and M2. During the transition phase tasks of both modes execute on the system.

[4] For the purpose of this thesis, the overhead involved with the execution of the mode change protocols is assumed negligible.

The real-time system is assumed to be composed of three CPUs which accommodate different applications depending on the operational mode. It is assumed that a mode change request MCR occurs at time t_{MCR} and imposes a mode change from M_1 to M_2 that consists in removing task τ_{1F} from CPU1 and adding tasks τ_{2A}, τ_{3A} on CPU1, and τ_{6A} on CPU2. The rest of the tasks τ_{4U}, τ_{5U} and τ_{7U} represent unchanged tasks and execute independent of the mode change. Because the transition between M_1 and M_2 is controlled by an asynchronous mode change protocol all seven tasks can (during the transition phase) simultaneously execute on the three CPUs of the system. It is further assumed that after a time interval Ψ after t_{MCR} task τ_{1F} is not executing anymore on CPU1 and the entire system executes in the steady mode M_2.

For such a system we are interested in bounding the duration of the transition phase and therewith in indicating the moment when the system reaches the steady state corresponding to the targeted operational mode (i.e. M_2). This is key for guaranteeing that the system can safely initiate a further transition (e.g. from M_2 to M_3) without the risk of overlapping with tasks on M_1.

For the purpose of this chapter the arbitration on the processing units is assumed to be performed according to the static-priority preemptive (SPP) policy. The tasks are ordered according to their priority, where τ_1 has the highest priority.

Each instance of a task τ_i, called a job and denoted with J_i, is activated by an event, which can be either external (such as interrupts) as in case of tasks τ_1, τ_2, τ_4 and τ_5 in Figure 4.1, or the result of another task or bus communication being finished (in which case there is a partial order between the possible task activations) as in case of task τ_3, τ_6 and τ_7. Tasks communicate via buffers. We assume that the task graph which describes the functional and timing dependencies between tasks does not contain cyclic dependencies. Functional dependencies are those dependencies given by the task graph (i.e. along the communication paths) and non-functional dependencies are those which arise from the local scheduling on a processor. As an example, a cyclic functional dependency in the system in Figure 4.1 would occur in case task τ_{3A} would trigger the execution of task τ_{2A} in addition to the external input. A cyclic timing dependency would occur in case task τ_2 and τ_3 would change positions.

Corresponding to the timing model in Section 2.2.2 task activation patterns are expressed with event streams using the upper and lower event arrival function $\eta_i^+(\Delta t)$ and $\eta_i^-(\Delta t)$ and the functions $\delta_i^+(n)$ and $\delta_i^-(n)$ which provide the maximum and the minimum number of events that occur in an event stream during any time interval of length Δt (see Figure 2.1). Each job of a task τ_i is further characterized by its worst-case execution time C_i and its (relative) deadline D_i, which may be smaller, equal, or larger than the distance to the successive activation. Jobs are executed in order, i.e. a new activated job will not execute before the previous job finishes.

For simplifying the explanations in the next sections, in Figure 4.1 tasks executing on the bus are not represented and will also not be further explicitly referred. However, the analysis method we contribute next accounts for the mode change effect on the communication medium whenever this is modelled similar to the processing units.

4.4 Bounding Mode Change Transition Latencies for Multi-Mode Real-Time Distributed Applications

4.4.1 The Mode Change Recurrent Effect: Problem Statement and Analysis Concepts

For the following explanations we consider the multi-mode system example depicted in Figure 4.1, where it is assumed that a mode change request (MCR) has imposed a configuration change on CPU1 and CPU2. In order to reason about specific points in time during the transition phase, we name the important points in time during a mode change and introduce metrics over time starting at the corresponding MCR.

As an example, we focus on the timing behavior of task τ_{4U} which has the lowest priority on CPU1. Figure 4.2a) depicts worst-case response time (WCRT) diagrams, which show the WCRT of tasks as a function of time after the temporal occurrence of the MCR. The upper diagram illustrates the transition effect on the timing behavior of task τ_{4U}. From the moment when the added tasks τ_{2A} and τ_{3A} are released on CPU1, i.e. at $t_{MCR}+\phi$, task τ_{4U} will experience additional interference and its WCRT will increase in comparison to the steady state before the MCR, i.e. $WCRT_{M1}^{\tau_{4U}} < WCRT_{Transition}^{\tau_{4U}}$. After task τ_{1F} finishes its execution corresponding to the activations released in the old mode it ceases to interfere the other lower priority local tasks, i.e. τ_{2A}, τ_{3A} and τ_{4U}. Thus, the WCRT of task τ_{4U} will decrease in comparison to the transition phase $WCRT_{Transition}^{\tau_{4U}} \geq WCRT_{M2}^{\tau_{4U}}$.

The WCRT values $WCRT_{M1}^{\tau_i}$ and $WCRT_{M2}^{\tau_i}$ for all tasks τ_i executing in the mutual exclusive execution modes M_1 and M_2 can be computed using existing analysis techniques as for example proposed in [154] and [65]. The WCRT during one transition phase, i.e. $WCRT_{Transition}^{\tau_i}$, can also be computed by assuming a *compound system* that includes all tasks executing in both operational modes, i.e. all unchanged, finished and new tasks in M_1 and M_2, as it was proposed in [65].

For the purpose of this chapter it is assumed that for all the tasks in a multi-mode system, the WCRT values for each individual mode and for each transition between two modes have been computed and are lower than the tasks deadlines such that the system is confirmed schedulable. Although this constitutes a conservative approximation of the system's behavior before, after and during the transition phase between two modes, it does not constitute a feasible approach if multiple mode changes (i.e. transition phases), for example from M_1 to M_2 and from M_2 to M_3, can overlap. In order to safely initiate another transition (e.g. from M_2 to M_3) the system must execute in the steady state corresponding to M_2. If a MCR that triggers the transition to a mode M_3 would be accepted before the previously initiated transition phase (i.e from M_1 to M_2) is finished, all the tasks in the system shall meet their deadlines in case of a compound system comprising tasks of three modes, e.g. M_1, M_2 and M_3. However, as illustrated with dashed line in Figure 4.2a), the WCRT of τ_{4U} would increase due to additional interference and thus τ_{4U} could miss it's deadline. Therefore, it is not enough to confirm the system schedulable in each operational mode and during every transition between

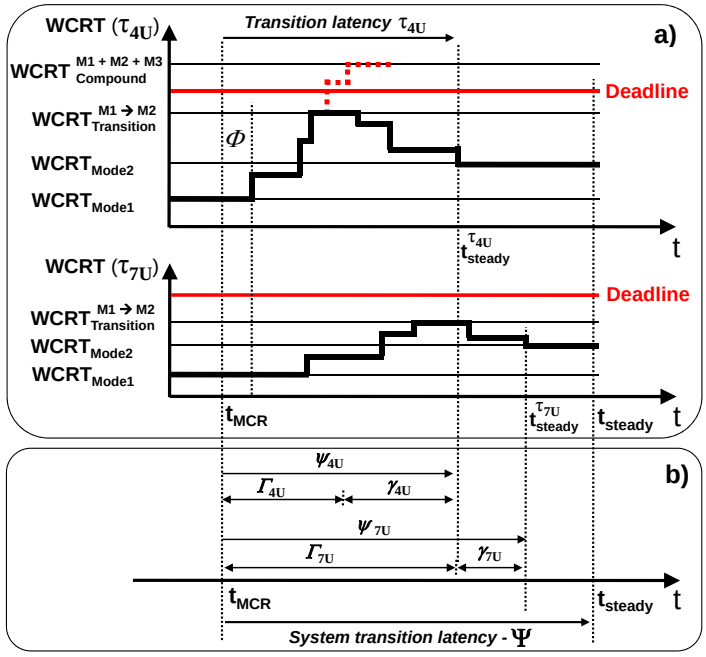


Figure 4.2: a) Illustration of a possible settling behavior for tasks τ_{4U} and τ_{7U}. b) Potential mode change time line for τ_{4U} and τ_{7U} in the context of the system transition latency.

two modes. The mode change transition latencies (i.e. the duration of the transition phases) have to be also computed in order to provide real-time guarantees for multi-mode real-time systems.

Similar to τ_{4U}, the timing behavior of other tasks in the system is certainly changing during the transition phase and will eventually settle at a time instant t^i_{steady} at which the $WCRT_{M2}$, corresponding to the new mode, can be safely assumed. As illustrated in Figure 4.2a), although the MCR is assumed to be a system-wide event and thus it marks a global point in time, timing effects may affect different tasks in the system for a different amount of time defined as follows.

Definition 4.1 ***(Task transition latency** ψ_i**)*** *The task transition latency ψ_i of a task τ_i is the maximum time that passes from the initiation of a mode change at t_{MCR} until the moment in time t^i_{steady} when all transient effects caused by the mode change ceased to affect the timing of this specific task.*

In order to make a system-wide decision on when a new mode change may be started without the risk of overlapping with the effects of a previous mode change, we do however need to compute the system transition latency.

Definition 4.2 ***(System transition latency** Ψ**)*** *The system transition latency Ψ is the maximum time that passes from t_{MCR} until a moment in time t_{steady} when all transient effects caused by the associated mode change ceased to affect the timing of all the tasks in the system.*

Thus, the largest transition latency ψ_i of any of the tasks running in the system represents the system transition latency

$$\Psi = max(\psi_i \mid \forall \tau_i) \tag{4.1}$$

and t_{steady} ($t_{steady} = t_{MCR} + \Psi$) indicates the latest point in time after the initiation of the mode change at t_{MCR} when the entire system reaches the steady state corresponding to the new mode.

The challenge of computing the system transition latency Ψ can be broken down to computing the tasks transition latencies ψ_i. However, in distributed systems, the task transition latency does not only depend on the tasks executing on the same processor but also on the other tasks in the system. For the multi-mode system example in Figure 4.1, when the mode change is initiated in the system, the execution of the finished task τ_{1F} may delay the execution of several jobs of task τ_{2A} activated after the MCR. This may lead to a transient overload situation which translates into a burst of events at the output of task τ_{2A} which then propagates to the input of task τ_{6A} on CPU2 and later to the input of task τ_{3A} on CPU1. From the perspective of task τ_{4U}, during the transition phase its jobs are delayed initially by the higher priority tasks τ_{1F} and τ_{2A}. This may lead to a burst of events at τ_{4U} output which propagates to the input of τ_{7U} on CPU3. After τ_{1F} completely finishes its execution and before the burst arrives at the input of τ_{3A}, task τ_{4U} is only delayed by the execution of task τ_{2A} which leads to a more relaxed output pattern of task τ_{4U} and therewith at the input of task τ_{7U}. When the burst of events arrives at the input of task τ_{3A}, task τ_{4U} will experience again increased interference from the higher priority tasks, which also means a possible new burst of activations at the input of τ_{7U}.

Thus, in distributed systems the effect of a mode change, i.e the transient overload caused by a MCR initiated at a time instant t_{MCR}, may be recurrent, propagating as waves through the system. This effect, propagating e.g. in form of burst of events, will arrive at the input port of the interconnected tasks, and therewith at the processor on which these tasks are mapped at a later time point which is defined as follows.

Definition 4.3 ***(Arrival of the mode change effect)*** *The arrival of a mode change effect at a resource indicates a moment in time relative to the initiation of a mode change at t_{MCR} when the effect of the mode change leads to a modification of the input activation pattern of any task mapped on that resource.*

For those resources (i.e individual CPUs or buses) on which the mode change imposes a configuration change such that tasks are added, removed or both, the *arrival of a mode change effect* coincides with the arrival of the MCR at time t_{MCR}. However, as the effect of a mode change propagates between the interconnected tasks, there are different and eventually multiple arrivals of the mode change effect at the input of different tasks mapped on the same or different resources - e.g. in the example above the effect of the mode change will propagate twice to the input of τ_{7U} even if on its host resource (i.e. on CPU3) there is no change imposed.

Therefore, for each task in a multi-mode system, the mode change timing has a local (resource-level) and a global (system-level) aspect. The transition latency ψ_i of a task τ_i, as defined in Definition 4.1, has two components, namely the *latency of the mode change effect* propagated by other tasks in the system to the resource on which τ_i is mapped, denoted with Γ_i, and the *task local transition latency*, denoted with γ_i.

Definition 4.4 ***(Task local transition latency** γ_i**)*** *The task local transition latency γ_i of a task τ_i is the maximum time that passes from the arrival of the mode change effect at the resource on which τ_i is mapped until the latest moment in time when this effect ceased to affect the timing of task τ_i on its local resource.*

Definition 4.5 ***(Task mode change effect latency** Γ_i**)*** *The mode change effect latency Γ_i of a task τ_i is the maximum time that passes from the initiation of a mode change at t_{MCR} until the moment in time when the effect of the mode change arrives for the last time at the resource on which τ_i is mapped.*

In the considered system, the mode change effect latency Γ_{7U} for task τ_{7U} represents the amount of time that passes from t_{MCR} until the second burst of activations at τ_{4U} output propagates to the input of τ_{7U}. A possible mode change time line for the tasks' transition latencies is depicted in Figure 4.2b).

In order to find the overall transition latency ψ_i of each task τ_i corresponding to Definition 4.1, one has to sum up the maximum local transition latencies γ_i and the mode change effect latencies Γ_i

$$\psi_i = \gamma_i + \Gamma_i \tag{4.2}$$

Thus, the problem of deriving the tasks transition latencies ψ_i, which is the main aspect of the next section, maps to the problem of computing upper-bounds for the parameters γ_i and Γ_i for all the tasks in the system. A solution for this is introduced in what follows.

4.4.2 Analysis of Mode Change Transition Latencies

4.4.2.1 Derivation of task local transition latency γ_i

In order to derive the worst-case transition latency analysis for the mode change model in Section 4.3 we rely on concepts used in the real-time scheduling theory.

For the calculation of the worst-case response time of a task τ_i scheduled on a single-core processor according to the static priority preemptive policy, one can rely on the *busy window* technique [78, 154]. In literature [154] (see also Definition 3.1) the *busy window of a task τ_i* (called also level-i busy window) is defined as the time interval for which a resource executes only tasks of priority greater than or equal to the priority of task τ_i and during which the resource is never idle. As discussed in Section 3.5.1 the *maximum busy window* of a task τ_i, denoted here with W_i^{max}, is obtained when jobs of task τ_i are assumed starting at the *critical instant* i.e. together with jobs of all the

higher priority local tasks. The maximum busy window of any task τ_i can be obtained by iteratively solving equation

$$w_i^{n+1}(q) = q \cdot C_i + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q)) \cdot C_j \tag{4.3}$$

where $w_i^n(q)$ is the maximum busy window of q activations of task τ_i with $q = 1, \ldots Q_i$ and $Q_i = \min\{q \geq 1 | w_i^n(q) < \delta_i^-(q+1)\}$, i.e. the iteration has to be continued as long as new activations of τ_i arrive before the previous finish; C_i is the WCET of a job of τ_i; $hpl(i)$ is the set of higher priority local tasks with τ_i; $\eta_j^+(w_i^n(q))$ provides the maximum number of jobs of tasks in $hpl(i)$ in a time window of size $w_i^n(q)$. A solution for (4.3) can be computed iteratively because all analysis components grow monotonically with respect to the window size. This means, the busy window analysis for processing resources scheduled according to static priority preemptive is order-preserving (see Theorem 3.1 and Corollary 3.2, which handles the more complex busy window equation that includes the blocking time analysis). In each iteration, the maximum workload of all tasks with priority higher than or equal to the priority of task τ_i is computed. Given the order-preservingness of the busy-window analysis procedure, the iterative computation stops either when two successive iterations provide identical values ($w_i^{n+1}(q) = w_i^n(q)$), or when some threshold (real-time constraint) is exceeded [154]. Finally, if the iterative calculation of (4.3) successfully finish, the maximum busy window we are interested in for any task τ_i is given by:

$$W_i^{max} = w_i(Q_i) \tag{4.4}$$

When a MCR imposes a configuration change on a resource such that tasks are added, removed or both, the equation for computing the maximum busy window has to be adapted to consider the execution of finished and added tasks. A key challenge is to identify, for each task τ_i under analysis, the worst-case scenario when the MCR shall occur such that it certainly leads to the worst-case execution during the transition phase.

Theorem 1 in [65] states and proofs conditions for identifying the *worst-case mode change scenario* (called also worst-case transition scenario) for a task τ_i on a single-core processor under static-priority preemptive scheduling. For clarity, we take over Theorem 1 from [65]:

Theorem 4.1 *The worst-case mode change scenario for a task τ_i on a single-core processor is obtained when t_{MCR} coincides with the activation instant of a finished higher priority local task in the set $hpl_F(i)$, all unchanged higher priority local tasks in the set $hpl_U(i)$ are released simultaneously with τ_i, i.e in the classical critical instant, and the added higher priority local tasks in $hpl_A(i)$ are arriving as early as possible after an offset ϕ after the initiation of the MCR.*

Three mode change scheduling examples for a task τ_i are depicted in Figure 4.3, 4.4 and 4.5 depending on the occurrence of the MCR.

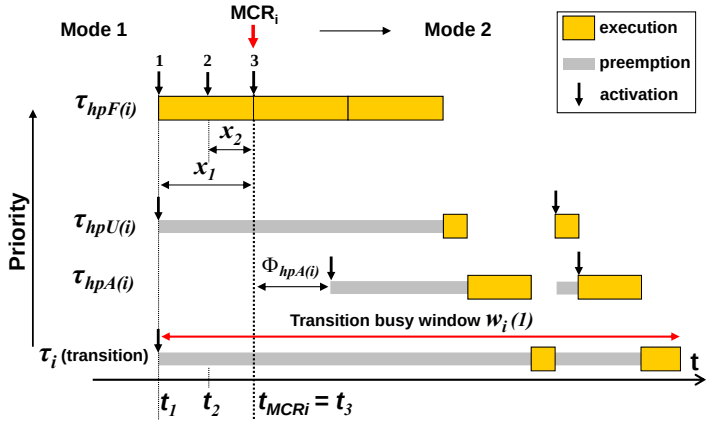


Figure 4.3: Scheduling example during a mode change where MCR_i coincides with the 3^{rd} activation of the finished task - **Worst-case mode change scenario**.

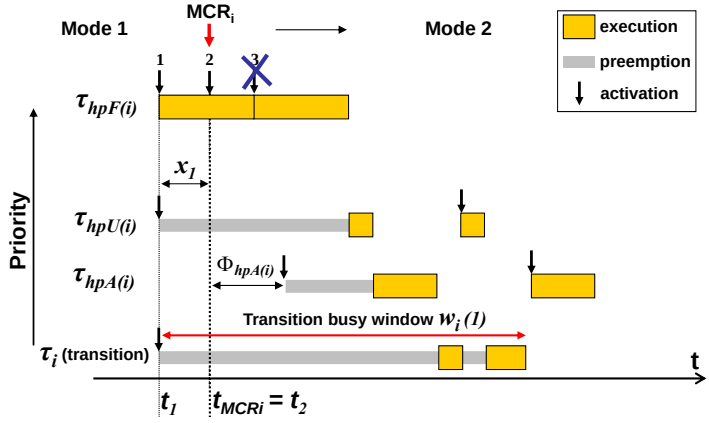


Figure 4.4: Scheduling example during a mode change where MCR_i coincides with the 2^{nd} activation of the finished task.

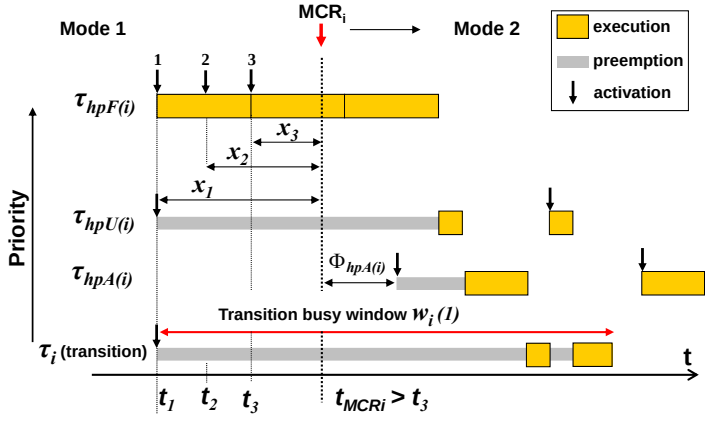


Figure 4.5: Scheduling example during a mode change where MCR_i occurs later than the 3^{rd} activation of the finished task.

The first scenario illustrated in Figure 4.3 considers the MCR occurring simultaneously to the third activation of the finished task. This scenario depicts the worst-case mode change scenario for task τ_i, where τ_i is delayed by all three activations of the higher priority finished task, by two activations of the higher priority unchanged task and by two activations of the higher priority added task.

In the second scenario it is assumed that the MCR coincides with the second activation of the finished task. As finished tasks are not activated after the MCR, the third activation of the higher priority finished task does not delay τ_i. Furthermore, the activations of the higher priority added task are moved earlier in time (their activation is relative to the MCR) fact that, combined with the reduced delay caused by the finished task, leads to a shorter busy window and therewith a faster completion of τ_i's execution.

In the third scenario, depicted in Figure 4.5, the arrival of the MCR is assumed to be later than the third activation of the higher priority finished task. In this case τ_i is delayed by all three activations of the finished task but as the activations of the added task are moved later in time τ_i experiences a better scenario in comparison to Figure 4.3, i.e τ_i is delayed only by the first activation of the added task.

According to Theorem 4.1 the MCR must be considered as coinciding with the activation instant of a finished higher priority task. However, there may be multiple higher priority finished tasks and for each of these tasks there may be several possible activations (i.e. jobs) released at different moments, as for example t_1, t_2 and t_3 in Figures 4.3 to 4.5 . Thus, one must identify all the time instances where the occurrence of the MCR should be assumed in order to find the worst-case transition scenario.

The moments in time corresponding to the activations of the higher priority finished tasks are relative to the occurrence of the MCR at t_{MCRi} (see Figure 4.3). Let X_i be the set of all possible time intervals x_i relative to t_{MCRi} which have to be investigated. Note that x_i essentially represents the transition busy window part before t_{MCRi}. The set X_i can be computed with Algorithm 1 presented in [65]. This is reproduced in the following.

Algorithm 4.1 Calculate X_i for the analyzed task τ_i

1: calculate a busy window L_i within the old mode scenario with a maximum workload of unchanged and finished tasks
2: **for** all $\tau_{jF} \in hep_F(i)$ **do**
3: /* $hep_F(i) = hpl_F(i) \bigcup \tau_i$, if τ_i is a finished task */
4: calculate $\eta^+_{jF}(L_i)$
5: **if** $\eta^+_{jF}(L_i) \geq 1$ **then**
6: **for** $n = 1$ to $\eta^+_{jF}(L_i)$ **do**
7: add $\delta^-_{jF}(n)$ to X_i
8: **end for**
9: **end if**
10: **end for**
11: remove duplicates from X_i

In the first line, Algorithm 4.1 calculates a busy window L_i within the old mode scenario with a maximum workload of unchanged and finished tasks. L_i can be obtained by iteratively solving the following equation [5]:

$$L_i^{n+1} = \sum_{\forall \tau_{jU} \in hpl_U(i)} \eta^+_{\tau_{jU}}(L_i^n) \cdot C_{jU} + \sum_{\forall \tau_{jF} \in hep_F(i)} \eta^+_{\tau_{jF}}(L_i^n) \cdot C_{jF} \tag{4.5}$$

Depending on its type, the analyzed task τ_i is either part of the set $hep_F(i)$ if itself is a finished task, or not considered if it is an added of unchanged task. If τ_i is an added task its execution is anyway not part of the old mode scenario and if τ_i is an unchanged task its activations are assumed simultaneously released (in critical instant) with all other higher priority unchanged tasks. Simply speaking, because for identifying the worst-case transition busy window for a task τ_i we are interested in the number and position of its higher priority finished tasks, we only need the maximum busy window of the finished task which has the lowest priority among all finished tasks with priorities above i. Those with priorities below i are not influencing the execution of τ_i across the mode change. If such a task does not exist, the transition busy window is constructed by starting at the classical critical instant scenario with the difference that added tasks are assumed released with an offset relative to the critical instant.

Thus, equation 4.5 is similar to 4.3 with the difference that unchanged and finished higher priority local tasks are explicitly captured by the different clauses. The first clause in 4.5 captures the maximum workload of higher priority unchanged tasks from $hpl_U(i)$ during L_i and the second clause calculates the maximum workload of higher priority finished tasks from $hep_F(i)$ during L_i. The calculation for 4.5 starts with an initial value $L_i(0) = 0$ and stops when two consecutive iterations provide identical values ($L_i^{n+1} = L_i^n$), or when some threshold (e.g. a real-time constraint) is exceeded.

In the lines 2 to 10, for each finished task τ_F from $hep_F(i)$ (it is including τ_i if this is a finished task), the Algorithm 1 calculates all possible values of x_i that have to be considered for constructing scenarios according to Theorem 4.1. This is done first by calculating the maximum activation number $\eta^+_{jF}(L_i)$ of τ_{jF} within L_i (line 4). For each of these activations, the algorithm calculates its corresponding value of x_i, i.e. the minimum distance between the busy window start and the activation occurrence (line 7). This distance can be calculated using the minimum distance function defined in Definition 2.3. Then, the calculated value is added to X_i. As different finished tasks may be activated simultaneously leading to identical values of x_i, the algorithm removes in line 11 the duplicates from X_i.

Having calculated all possible values of x_i, the busy window for τ_i have to be calculated for each x_i. The largest busy window obtained for any of the values x_i represents the task *maximum busy window* of a task τ_i during which a MCR occurs [65].

The maximum busy window of a task τ_i can be calculated by iteratively solving equation (4.6) if τ_i is an unchanged or finished task and equation (4.7) if τ_i is an added

[5] The iterative calculation is possible as all components of the busy window analysis for SPP scheduling are order preserving - see the more complex setups covered by Theorem 3.1 and Corollary 3.2.

task. The clauses in equation (4.6) and (4.7) consider the maximum workload due to execution of unchanged, finished and added tasks with priority higher than or equal to the priority of task τ_i.

$$\begin{aligned} w_i^{n+1}(q) = q \cdot C_i + & \\ & \sum_{\forall \tau_{jU} \in hpl_U(i)} \eta^+_{\tau_{jU}}(w_i^n(q)) \cdot C_{jU} + \\ & \sum_{\forall \tau_{jF} \in hpl_F(i)} \eta^+_{\tau_{jF}}(x_i) \cdot C_{jF} + \\ & \sum_{\forall \tau_{jA} \in hpl_A(i)} \eta^+_{\tau_{jA}}(w_i^n(q) - x_i - \phi_{\tau_{jA}})_0 \cdot C_{jA} \end{aligned} \tag{4.6}$$

$$\begin{aligned} w_i^{n+1}(q) = min(q \cdot C_i, \eta_i^+(w_i^n(q) - x_i - \phi_i)_0) + & \\ & \sum_{\forall \tau_{jU} \in hpl_U(i)} \eta^+_{\tau_{jU}}(w_i^n(q)) \cdot C_{jU} + \\ & \sum_{\forall \tau_{jF} \in hpl_F(i)} \eta^+_{\tau_{jF}}(x_i) \cdot C_{jF} + \\ & \sum_{\forall \tau_{jA} \in hpl_A(i)} \eta^+_{\tau_{jA}}(w_i^n(q) - x_i - \phi_{\tau_{jA}})_0 \cdot C_{jA} \end{aligned} \tag{4.7}$$

Equation (4.6) has to be used if the analyzed task τ_i is an unchanged (τ_{iU}) or a finished task (τ_{iF}). The only difference between the calculation of the worst-case response-time for finished and unchanged tasks with (4.6) is given by the termination of the iterative calculation. When analyzing an unchanged task τ_{iU} the iteration is performed for all jobs $q = 1, \dots Q_i$ with $Q_i = \min\{q \geq 1 | w_i^n(q) < \delta_i^-(q+1)\}$. In other words, the iteration has to be continued as long as new activations of τ_{iU} arrive before the previous finish. For a finished task τ_{iF} one has to iterate only over those jobs of τ_{iF} which are activated within x_i, i.e. only for those jobs which are activated before the occurrence of the MCR. This means the calculation is performed for all jobs $q = 1, \dots Q_i$ with $Q_i = \min\{q \geq 1 | w_i^n(q) < \delta_i^-(q+1)$ && $q \leq \eta_i^+(x_i)\}$.

Equation (4.7) is similar to (4.6), but considers that the analysed task τ_i is an added task which can not be activated before $\phi_i + x_i$ time units after the start of the transition busy window. The first clause in equation (4.7) indicates that, for large values of the offset ϕ_i, task τ_i does not contribute to the busy window $w_i(q)$. The function $\eta^+_{\tau_A}(w_i(q) - x_i - \phi_{\tau_A})_0$, which indicates the maximum number of higher priority added tasks that can interfere with the execution of the analyzed task τ_i, represents a modified version of the original upper event arrival function $\eta^+(\Delta t)$ and returns 0 if $w_i(q) - x_i - \phi_i < 0$.

For each arrival of a mode change effect to a resource (see Definition 4.3), the maximum busy window W_i^{max} of any task τ_i in (4.4) is obtained:

(i) with (4.3) by assuming the classical critical instant scenario in case the mode change effect only modifies the input activation pattern of the tasks without changing the set of tasks executing on that resource and

(ii) with (4.6) or (4.7) for all identified values x_i with Algorithm 4.1 in case the mode change effect is caused by a configuration change of the task set on that resource.

Relying on the busy windows characteristics mentioned above and proven in the literature [78, 154, 65], the maximum busy window W_i^{max} of a task τ_i in a multi-mode system represents the longest time interval required by jobs of this task to complete their execution affected by the arrival of a mode change effect.

Theorem 4.2 *For each arrival of a mode change effect, the local transition latency γ_i of a task τ_i is upper bounded by the task maximum busy window computed for the configuration of the input activation pattern of the tasks corresponding to the mode change arrival.*

$$\gamma_i \leq W_i^{max} \tag{4.8}$$

Proof: The proof follows by contradiction. Let's assume there is a time interval W ($W > W_i^{max}$) required by jobs of τ_i to finish their execution corresponding to the same configuration of the tasks' input activation pattern (given by event streams represented by the functions $\eta^+(\Delta t)$) as used when computing the maximum busy window W_i^{max}. This means, there is a time interval starting at a time instant other than the *critical instant* or the *worst-case mode change scenario*, in which the workload of tasks of priority greater than or equal to the priority of task τ_i is larger than the workload of the same tasks computed in the maximum busy window. This contradicts the assumptions and the definition of the maximum busy window. □

4.4.2.2 Computation of the system transition latency Ψ

In Section 4.4.2.1 we showed that the local transition latencies γ_i of any task τ_i is upper-bounded by the maximum transition busy window - with (4.8) - computed under the worst-case mode change assumptions for any input event models at the input of the local tasks with τ_i. The computation of the worst-case transition busy windows in multi-mode distributed systems with communicating tasks can be performed, for example, with the compositional analysis methodology in [64], where task activating event models are provided and iteratively refined during the worst-case system-level analysis procedure.

Having for all tasks on all processors the largest possible local transition latencies γ_i, these can be used for computing the mode change effect latencies Γ_i and therewith the transition latencies ψ_i with (4.9) and (4.10) as presented in Section 4.4.2.3.

Further, the system transition latency Ψ is upper-bounded by the largest task transition latency ψ_i of any task in the system with (4.1) and the latest moment in time when the system definitely reaches the steady state corresponding to the new mode relative to the occurrence of a MCR at t_{MCR} is given by $t_{steady} = t_{MCR} + \Psi$.

As an interesting result, the computation of the mode change effect latencies Γ can not be mapped to the seemingly related problem of computing end-to-end delays as presented for example in [152]. Besides the fact that existing end-to-end approaches were not developed for multi-mode setups, they compute only the largest end-to-end delay of one activation. As seen above the propagation of the mode change effect comprises

multiple activations which can be captured only by the busy windows. Furthermore, end-to-end approaches do not cover the effects that propagate through non-functional dependencies. For example, when computing the end-to-end delay from task τ_4 to τ_7 in the timing dependency graph in Figure 4.6 the influence of τ_{6A}'s execution on the execution of tasks τ_4 and τ_7 is not captured.

4.4.2.3 Derivation of the mode change effect latency Γ_i

As discussed in Section 4.4.1, the recurrent effect of a mode change propagates through a multi-mode system and affects the transition latency ψ_i of a task τ_i through the mode change effect latency Γ_i (see Definition 4.1 and relation (4.2)). In other words, the transition latency ψ_i of a task τ_i does not only depend on its worst-case execution and worst-case interference on its local resource, i.e. γ_i, but also on the latency of the mode change effect Γ_i propagated by other tasks, possibly mapped on other resources.

In order to derive the mode change effect latencies in multi-mode distributed systems, we integrate the local resource-level (i.e processor) timing view into a global system-level timing view. In Figure 4.6 we introduce a *timing dependency graph* which indicates the functional and non-functional dependencies between the tasks in the system in Figure 4.1.

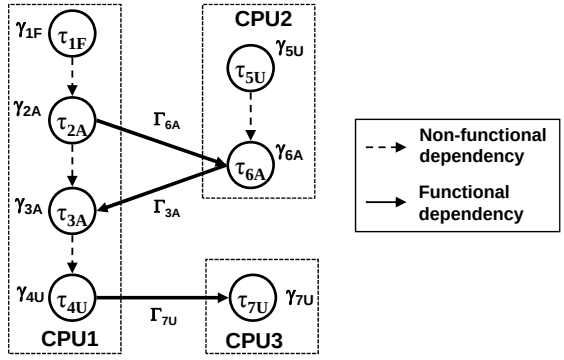


Figure 4.6: Timing dependency graph for the system example in Figure 4.1.

The nodes of the graph correspond to the tasks in the system and the directed edges represent functional and non-functional dependencies between tasks. Functional dependencies are those dependencies given through the task graph and non-functional dependencies are those which arise from the local scheduling on a processor. The direction of the edges in Figure 4.6 indicates the direction of influence between the tasks. Remember that for the purpose of this paper, we don't consider systems for which the timing dependency graph contains cyclic dependencies.

The nodes of the graph in Figure 4.6 are annotated with the values γ_i corresponding to the largest tasks local transition latencies obtained with (4.8) in Theorem 4.2 (i.e. $\gamma_i = W_i^{max}$). The edges which correspond to the functional dependencies between tasks are annotated with the values Γ_i. These indicate that the effects of a mode change propagate to the input ports of the functionally interconnected tasks. For each task in the system, we are interested in upper bounding the time interval Γ_i which starts at

t_{MCR}. The end of the time interval Γ_i, will indicate the latest point in time after which the mode change effect won't be further propagated to the input port of a task by the functionally interconnected tasks.

Thus, the activating task of a task τ_i, denoted with τ_i^p, which is the immediate predecessor of task τ_i in the task graph (indicated with solid lines in Figure 4.6), will propagate the mode change effect to the input of task τ_i (e.g. $\tau_{6A}^p = \tau_{2A}$). For the next explanations we denote with $\mathcal{T}_{hep(i)}$ the set of tasks which contains task τ_i and the other local tasks with priorities higher than the priority of τ_i. Further, we denote with $\mathcal{T}_{hep(i)}^p$ the set of tasks which are the immediate predecessors of tasks in $\mathcal{T}_{hep(i)}$ in the tasks graph. Tasks in $\mathcal{T}_{hep(i)}^p$ have direct functional dependencies with the tasks in $\mathcal{T}_{hep(i)}$. For example, for task τ_{4U} in the timing dependency graph in Figure 4.6 we have $\mathcal{T}_{hep(4U)} = \{\tau_{1F}, \tau_{2A}, \tau_{3A}, \tau_{4U}\}$ and $\mathcal{T}_{hep(4U)}^p = \{\tau_{6A}\}$.

Theorem 4.3 *The only tasks that can propagate the effect of a mode change to the input port of a task τ_i are the activating tasks of all tasks with priority higher than or equal to the priority of task τ_i, this means tasks in $\mathcal{T}_{hep(i)}^p$.*

Proof: In Section 4.4.2.1 it was proven that the local transition latency γ_i (i.e maximum transition busy window) of a task τ_i only depends on the execution of tasks with priorities higher and equal to the priority of task τ_i, i.e. tasks in $\mathcal{T}_{hep(i)}$. A modification of the input activation pattern (given by the input event stream represented by the functions $\eta^+(\Delta t)$) of the tasks in $\mathcal{T}_{hep(i)}$ will modify the local transition latency γ_i of task τ_i. In case of communicating tasks, the inputs of tasks in $\mathcal{T}_{hep(i)}$ are connected to their immediate predecessors in the task graph, which are the tasks in $\mathcal{T}_{hep(i)}^p$. □

Corollary 4.4 (Stopping condition for mode change effect propagation) *The mode change effect will not be further propagated to a task τ_i after the moment in time when the mode change effect ceased to affect the timing of all the tasks that can propagate this effect to task τ_i, i.e. the timing of all the tasks in $\mathcal{T}_{hep(i)}^p$.*

From Definition 4.1, the mode change effect ceases to affect the timing of a task τ_i at the end of its transition latency ψ_i. Thus, the mode change effect latency Γ_i of a task τ_i is a function of the transition latencies ψ_j of the tasks $\tau_j \in \mathcal{T}_{hep(i)}^p$ which can propagate the effect of a mode change to the input of a task τ_i.

Theorem 4.5 *The mode change effect latency Γ_i of a task τ_i is upper bounded by the maximum task transition latency ψ_j over all tasks that can propagate the mode change effect to the input port of a task τ_i, if any.*

$$\Gamma_i \leq max(\psi_j, 0), \forall \tau_j \in T_{hep(i)}^p \tag{4.9}$$

Proof: On one hand, if the set $\mathcal{T}_{hep(i)}^p$ is not empty, the task transition latency ψ_j for each task τ_j in this set has to be computed. The maximum of all transition latencies

ψ_j indicates the latest moment in time relative to the initiation of a mode change at t_{MCR} when the timing of a task that can propagate the mode change effect ceased to be affected. After this moment in time, none of the tasks in $\mathcal{T}^p_{hep(i)}$ will further propagate the effect to τ_i.

On the other hand, from Theorem 4.3 we know that if for any task τ_i there is no task $\tau_j \in \mathcal{T}^p_{hep(i)}$, then the mode change effect does not propagate to τ_i and the mode change effect latency Γ_i is 0. □

The problem of deriving upper bounds for the task transition latencies ψ_i of each task τ_i in the system is recurrent, because ψ_i requires upper bounds of Γ_i which in turn requires upper bounds of the task transition latencies ψ_j of the tasks $\tau_j \in \mathcal{T}^p_{hep(i)}$ (Theorem 4.5). Thus, it has to be proven that the task transition latency ψ of each task in the system is upper bounded by $\gamma + \Gamma$ in (4.2).

Theorem 4.6 *For each task τ_i in a multi-mode distributed system without cycles in the timing dependency graph the task transition latency ψ_i is upper bounded by*

$$\psi_i \leq \gamma_i + \Gamma_i \tag{4.10}$$

Proof: The proof is by contradiction and induction along the timing dependency graph. Let's assume for task τ_i there is a time interval $\tilde{\psi}_i$ such that $\tilde{\psi}_i > \psi_i$. This means

(i) $$\exists \tilde{\psi}_i : \tilde{\psi}_i > \psi_i \Longrightarrow \exists \tilde{\gamma}_i, \tilde{\Gamma}_i : \tilde{\gamma}_i + \tilde{\Gamma}_i > \gamma_i + \Gamma_i$$

From Theorem 4.2 we know there is no $\tilde{\gamma}_i > \gamma_i$. Thus, the problem reduces to

(ii) $$\tilde{\Gamma}_i > \Gamma_i$$

From Theorem 4.5, $\Gamma_i \leq max(\psi_j, 0), \forall \tau_j \in T^p_{hep(i)}$. By replacing Γ_i in (ii) we have

(iii) $$max(\tilde{\psi}_j, 0) > max(\psi_j, 0), \forall \tau_j \in T^p_{hep(i)}$$

This leads to the initial problem $\tilde{\psi}_j > \psi_j$ in (i) and further to (ii) $\tilde{\Gamma}_j > \Gamma_j, \forall \tau_j \in T^p_{hep(i)}$.

By applying Theorem 4.5 the problem follows along the dependency graph for each task $\tau_j \in T^p_{hep(i)}$ and further for each $\tau_k \in T^p_{hep(j)}$ until a task τ_x, for which $T^p_{hep(x)} = \emptyset$ such that $\Gamma_x = 0$. However, for any graph without cyclic dependencies $\exists \tau_x : T^p_{hep(x)} = \emptyset$ and thus $\tilde{\Gamma}_x = 0$, from which

$$\Rightarrow \exists \tau_x : \tilde{\Gamma}_x = \Gamma_x, \text{ which contradicts } \tilde{\Gamma}_x > \Gamma_x.$$

Having the tasks τ_x, for which $\tilde{\Gamma}_x = \Gamma_x = 0$, as the base of induction, the inductive steps follow for all the tasks along the dependency graph and contradict the assumption $\tilde{\Gamma}_i > \Gamma_i$ for each task in the system [6]. □

[6] This means that (4.9) and (4.2) are applied for all the nodes (i.e. tasks) that can be reached by

4.4.2.4 Example of the analysis procedure

In this section we make use of the system example in Figure 4.1 with the timing dependency graph in Figure 4.6 to describe the analysis procedure introduced above. We focus on the analysis of task τ_{7U} as this is the more complex case the analysis procedure has to solve for this example.

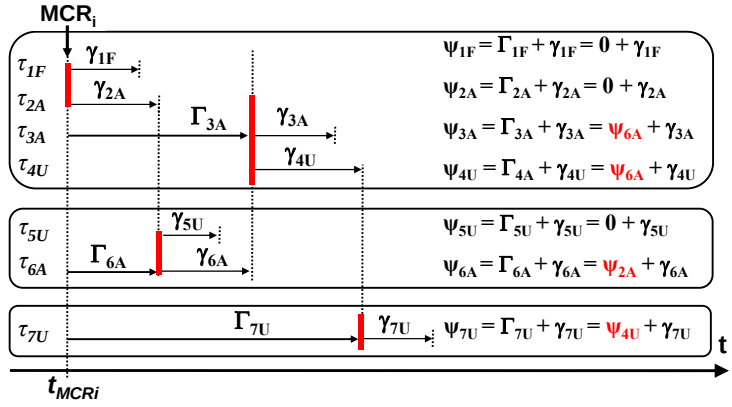


Figure 4.7: Task transition latencies in the context of the system transition phase.

In Figure 4.7 we introduce a mode change time line which depicts the task transition latencies in the context of the system-level mode change transition phase. The vertical bold lines, illustrate for each task the latest moment in time when the mode change effects are propagated for the last time at their input.

In order to compute the task transition latency ψ_{7U}, upper bounds on the local transition latency γ_{7U} and on the mode change effect latency Γ_{7U} have to be derived. The local transition latency γ_{7U} is upper-bounded by the maximum transition busy window corresponding to Theorem 4.2. As τ_{7U} is activated by τ_{4U} the mode change effect latency Γ_{7U} is a function of the task transition latency ψ_{4U} of task τ_{4U}, which in turn is a function of γ_{4U} and Γ_{4U}. By applying (4.9) and (4.10) (see Section 4.4.2.3) the computation of the task transition latency of task τ_{7U} is performed as given below:

$$\begin{aligned}\psi_{7U} &= \Gamma_{7U} + \gamma_{7U}, \;\; \Gamma_{7U} = max(\psi_{4U}, 0) \\ &= \psi_{4U} + \gamma_{7U} \\ &= \Gamma_{4U} + \gamma_{4U} + \gamma_{7U}, \;\; \Gamma_{4U} = max(\psi_{6A}, 0) \\ &= \psi_{6A} + \gamma_{4U} + \gamma_{7U}\end{aligned}$$

performing a backwards search over the edges corresponding to the functional and non-functional dependencies in the timing dependency graph. In a task graph without cyclic dependencies, the backwards search over paths will reach starting nodes in a finite number of steps x. For the computation of the mode change effect latency, finding starting nodes means that there is at least one task τ_x for which the task transition latency ψ_x does not depend on other tasks such that $\Gamma_x = 0$. Starting from these tasks, for which the task transition latency ψ_x is given only by their local transition latency γ_x (i.e. $\psi_x = \gamma_x + 0$), the computation of the mode change effect latencies Γ of other tasks can be performed straightforwardly.

$$= \Gamma_{6A} + \gamma_{6A} + \gamma_{4U} + \gamma_{7U}, \;\; \Gamma_{6A} = max(\psi_{2A}, 0)$$
$$= \psi_{2A} + \gamma_{6A} + \gamma_{4U} + \gamma_{7U}$$
$$= \Gamma_{2A} + \gamma_{2A} + \gamma_{6A} + \gamma_{4U} + \gamma_{7U}$$
$$(\Gamma_{2A} = 0 \text{ as } T^{p}_{hep(2A)} = \emptyset)$$
$$\psi_{7U} = \gamma_{2A} + \gamma_{6A} + \gamma_{4U} + \gamma_{7U}$$

Similarly, for each task in the multi-mode distributed system without cyclic dependencies the task transition latency can be computed in a finite number of steps.

4.4.3 Experiments

In this section we show the applicability of the proposed approach. For this, we consider the system example depicted in Figure 4.1 that undergoes a mode change so that task τ_{1F} is removed from CPU1 and tasks τ_{2A}, τ_{3A} and task τ_{6A} are added on CPU1 and CPU2 respectively. For this system we assume the parameters given in Table 4.1.

Table 4.1: Parameters for the system in Figure 4.1

Mapping	Task Name	Execution Mode	Activation Source	Priority	Activation Period T_i (ms)	WCET C_i (ms)
CPU1	τ_{1F}	1	I_1	1	120	12
CPU1	τ_{2A}	2	I_2	2	16	2
CPU1	τ_{3A}	2	τ_{6A}	3	(*	3
CPU1	τ_{4U}	1 and 2	I_3	4	11	3
CPU2	τ_{5U}	1 and 2	I_4	5	170	33
CPU2	τ_{6A}	2	τ_{2A}	6	(*	2
CPU3	τ_{7U}	1 and 2	τ_{4U}	7	(*	4

(* - tasks are event driven activated by predecessors

Considering these parameters, the utilization of CPU1 in mode M_1, before the MCR, is about 37%. During the transition phase, as the finished task τ_{1F} and the added tasks τ_{2A} and τ_{3A} may interfere, the utilization increases to 68.5%. In the steady state corresponding to mode M_2, when τ_{1F} is not executed anymore, the utilization of CPU1 decreases to 58.5%. Thus, we consider the case when CPU1 will switch from a lower CPU utilization level in mode M_1 to a higher CPU utilization level in mode M_2.

The results of the analysis for the individual task transition latencies and for the system transition latency in this setup are presented in Table 4.2. In the worst-case situation the system settles 138 ms after the initiation of the considered mode change, far later than the response time of any of the tasks in the system due to the "wave" effect. After this time interval the system can be assumed executing in the steady state M_2, and a new mode change can be safely started without the risk of overlapping with the effects of the previous mode change, i.e. from M_1 to M_2.

In the next experiment we deviate from the periodic assumptions and increase the jitter of τ_{1F} activations. With this, we analyse the case where a MCR occurs at different moments in time when the backlog in the input buffer of τ_{1F} is large and has to be

Table 4.2: Analysis results: Task and system transition latencies

Transition Latency	Tasks						
	τ_{1F}	τ_{2A}	τ_{3A}	τ_{4U}	τ_{5U}	τ_{6A}	τ_{7U}
γ_i	12	14	31	59	33	41	24
Γ_i	-	-	$\psi_{6A} = 55$	$\psi_{6A} = 55$	-	$\psi_{2A} = 14$	$\psi_{4U} = 114$
ψ_i	12	14	86	114	33	55	138
Ψ	138						

executed during the transition phase. The execution of multiple instances of the finished task during the transition phase leads to increased interference on the other local tasks and therewith to increased system settling times, i.e. system transition latencies.

The system transition latencies depending on the activation backlog of task τ_{1F} are depicted with triangles in Figure 4.8. Further, by modifying the activation period of the

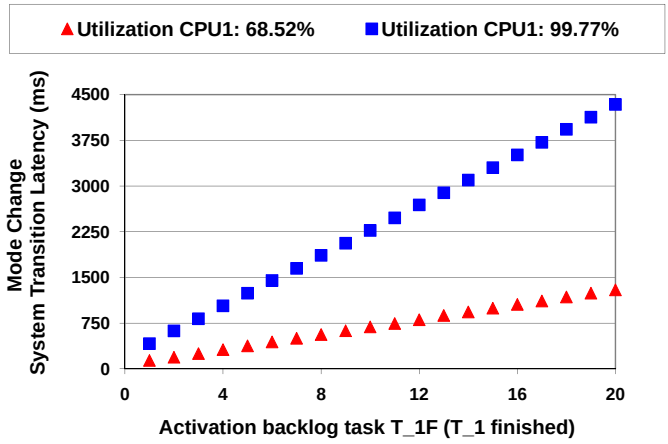


Figure 4.8: Mode change system transition latencies depending on the activation backlog of the finished task τ_{1F}.

added task τ_{2A} mapped on CPU1, we increased the utilization level of CPU1 corresponding to M_2 to about 89%. As an effect, the utilization of CPU1 during the transition phase approached 99.77%. Similar to the previous experiment, we varied the activation backlog of the finished task τ_{1F}. The resulting mode change system transition latencies, depicted with rectangles in Figure 4.8, indicate the significant impact of the increased CPU utilization on the system settling behavior.

We repeated the experiments described above for the case when task τ_{1F} is an unchanged task, i.e. $\tau_{1F} = \tau_{1U}$. With this we considered the case where the functionality of a system is extended with a new application composed of the tasks τ_{2A}, τ_{3A} and τ_{6A}. In this setup, the execution of lower priority added and unchanged tasks on CPU1 will be interfered not only by the activations of τ_{1U} pending when the MCR occurs, but also by its next activations released after the mode change initiation. As can be seen in Figure 4.9, this prolongs the settling time of the mode change effects. For the case with

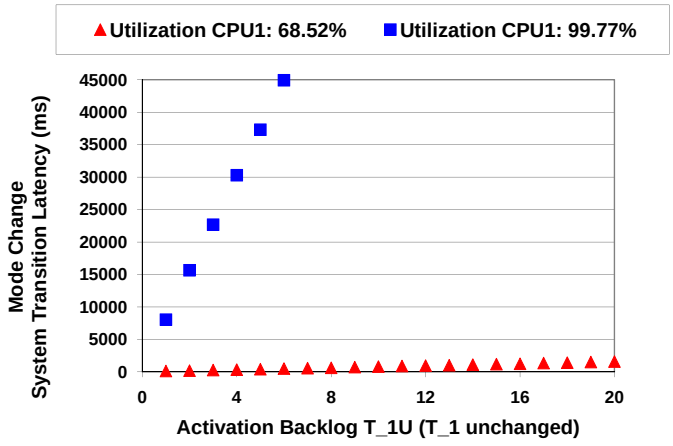


Figure 4.9: Mode change system transition latencies depending on the activation backlog of task τ_{1F} modelled as unchanged task.

CPU utilization level approaching 99.77%, the system transition latencies dramatically rise already for few pending activations at the mode change initiation.

4.4.4 Case Study

In what follows, we introduce an automotive specific use case in order to explain and exemplify how the formal analysis method introduced earlier in this section can be applied in the current automotive practice.

4.4.4.1 System Model of an Automotive System

For the next explanations we refer to the system depicted in Figure 4.10 which abstracts a partitioned multiprocessor system with two cores independently scheduled according to a fixed-priority scheduler (e.g. OSEK/VDX [100]). The elements of the system in Figure 4.10 mainly corresponds to the system and mode change model introduced in Section 4.3. The system consists of several tasks characterized by their priorities (given

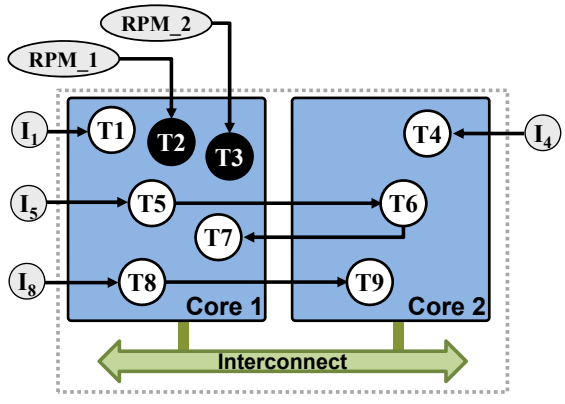


Figure 4.10: Illustration of a dual-core processor with inter-core communication.

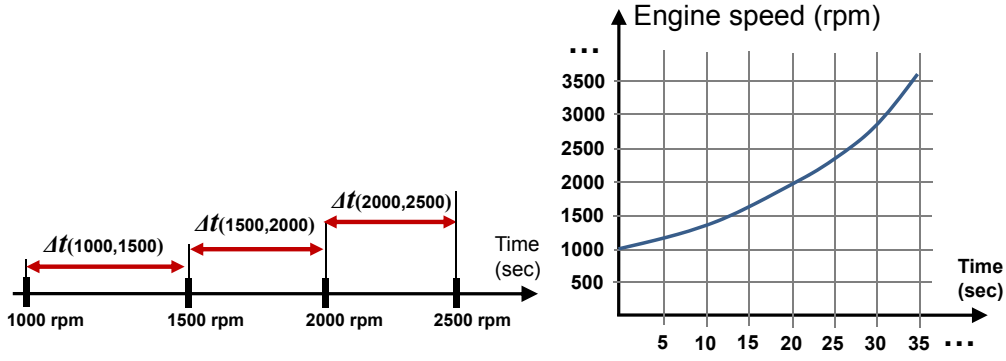
by the index), their execution times, their activation periods and their deadlines, which may be smaller, equal, or larger than the periods. The activation of a task is triggered by an activating event, which may be the result of timer expiration, an external or internal interrupt (I1, I4, I5 and I8 in Figure 4 represent the event sources at the task input), or the result of another task being finished.

With task T2 and T3 we modell so-called "engine-synchronous" tasks, a special type of periodic tasks specific to automotive powertrain controllers. Such tasks measure the engine state and control actuators such as fuel injection several times per engine rotation. Thus, the activation of T2 and T3 is given by the engine speed, measured in revolutions per minute (rpm). The recurrence of the engine-synchronous tasks depends on the camshaft and crankshaft positions that vary with the engine speed and is therefore expressed in engine angle degree rather than time. For example, lets assume that task T2 in Figure 4 is activated each 90^0 (i.e. four time per rotation) and task T3 each 360^0 (i.e. once per rotation). In order to obtain a system-wide unique time base, for each fixed engine speed at which an engine-synchronous task is to be specified, the angular recurrence has be transformed in time units. The activation periods of the tasks T2 and T3 at different fixed engine speed values are given in Table 4.3.

Table 4.3: Parameters of engine synchronous tasks

Period	Time Units (ms) at constant engine speeds (RPM)										
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000
P2	15	10	7.5	6	5	4.28	3.75	3.33	3	2.75	2.5
P3	60	40	30	24	20	17.14	15	13.33	12	10.9	10

In order to capture more exactly the behavior of engine-synchronous tasks, the time duration an engine needs to accelerate or decelerate between two discrete engine speed values rpm1 and rpm2 can be modelled with a time interval Δt(rpm1, rpm2) (see Figure 4.11a). In practice, these time intervals depend on the gear, on the current cruising



(a) (b)

Figure 4.11: a) Time intervals between two constant engine-speed values. b) Example of engine-speed variation over time during an acceleration phase.

speed and the driving behavior, and can be obtained e.g. by analysing the acceleration behavior of a car using test benches [109] or by relying on real field tests. The results in [109] indicate for a particular setup an acceleration phase of 20sec from 1000rpm to 3000rpm in the 4th gear and of 35sec in the 5th gear. Figure 4.11b depicts a possible scenario during acceleration.

4.4.4.2 Problem Statement

As shown by Symtavision GmbH [147] in [91], with the increasing engine speed, the load on engine control units cores increases due to the higher rate of task activations. Figure 4.12 [7] illustrates a possible load situation for a dual-core system as modelled in Figure 4.10. The diagram in Figure 4.12 captures the task workload that has to be serviced by each core, ignoring any inter-core communication effects.

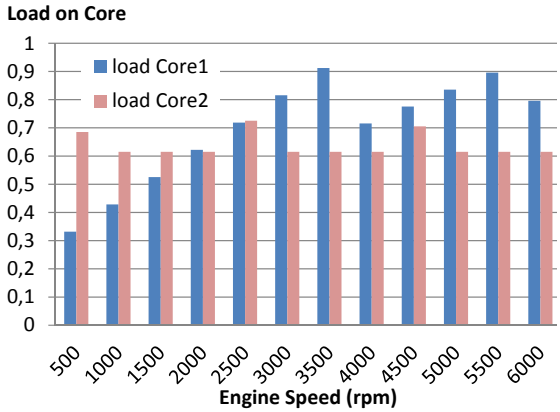


Figure 4.12: Workload to be processed on each core. Increasing engine speed leads to higher rate of activation for engine-synchronous tasks. In addition task modes lead to varied workload. Critical load on Core 1 is reached around 3500 and 5500 rpm.

If no further measures are in place, the load may eventually increase above a critical value, making the system unschedulable and the engine control inefficient or even unstable. Different solutions are therefore applied in order to reduce the workload at high engine speeds. For example, some tasks are changing their behavior at higher engine speeds, computing only rough control values (i.e. tasks reduce their execution times). Furthermore, other tasks are aborted when the engine speed reaches a critical value. In this way, automotive power train controllers resort to mode change mechanisms in order to perform at a high quality for low engine speeds and to adequately operate also under high engine speeds.

This behavior can be reflected in a scheduling model of the system. Lets assume that starting e.g. at 3500rpm, the engine-synchronous tasks change their behavior and some

[7] The diagram in Figure 4.12 reproduce the figure provided by Symtavision GmbH for [91]. This diagram can be easily produced with the model based scheduling analysis tool SymTA/S [147].

internal functions are completely shut-off (e.g. "high-quality mode" to medium quality mode). Later e.g. at 5500rpm, additional functions are turned off (e.g. "low quality mode"). In addition, some functions provided by the periodic tasks are reactive to the current processor load. This task flexibility ensures that the schedulability is maintained through the complete spectrum of operating conditions.

In Figure 4.12 one can see the effect of changed task behavior on Core 1 at engine speeds 3500rpm and 5500rpm, where task T2 is put into a reduced quality mode. In addition, also some periodic tasks in this example have increased execution time requirements at certain engine speeds (which leads to a slight load variation on Core 2 at 2500 and 4500 rpm).

As can be seen, a simple and fast way to treat critical situations in the industrial practice (e.g. in case of increased load at high engine speeds) is to just abort some of the processed system functions in order to permit the safe execution of critical functions. However, even if by implementing severe solutions the safe system functionality can be ensured (e.g. by aborting tasks or by reducing their computational requirements), the resulting service degradation is not convenient and will become unacceptable with the increasing requirements for lower emissions and continued demands for improved fuel economy. When functions are suddenly aborted, just like in case of processor interrupts, data are lost. Instead of executing such sharp transitions, gradual mode transitions are preferable such that functions can be resumed and efficiently continued later.

4.4.4.3 New Design Options For Automotive Multi-Mode Applications

The variable recurrence of the engine-synchronous tasks at runtime leads to a continuous change in the configurations that have to be taken into account for the OS schedule on the cores and leads to a multi-mode behavior of the entire system. As discussed in Section 4.4.4.2 the methodology available for timing and performance design can be applied to real-time systems which accommodate tasks with angular recurrence (for more details see [91]). However, these methods only suit the current automotive practice, where overly pessimistic measures are applied in order to permit the safe system functionality in critical situations (e.g. by aborting tasks or by reducing their computational requirements in case of increased processor load at high engine speeds).

Based on the modeling and analysis approach introduced in Section 4.3, 4.4.1 and 4.4.2 we propose to map the problem of scheduling real-time applications which accommodate tasks with angular recurrence to the problem of scheduling multi-mode applications. Relying on this, we next discuss new options for the design and analysis of automotive specific multi-mode systems.

4.4.4.3.1 Mode Change Model applied to the Automotive Case Study

Based on the general modeling solution introduced in Section 4.3, the multi-mode behavior of the system considered in this use case can be captured as follows. The different operational modes of the system depicted in Figure 4.10 can be specified by a finite set $M = \{M1, M2, \ldots, Mx\}$. Each mode $M_i (M_i \in M)$ is characterized by a

different behavior and is associated with a specific set of tasks. e.g. in mode $M1$ we have all nine tasks running in the system, in $M2$ we have only tasks T1 to T7, and finally in $M3$ we have only the task T1 to T4. In response to a mode change request (MCR), initiated by a system internal event at a certain engine speed value, the multi-mode systems will experience transitions from an old operational mode characterized by a set of functionalities, to a new operational mode characterized by a different or a changed set of functionalities as follows. At a certain engine speed a mode change request (MCR1) triggers the transition from the operational mode $M1$ to the operational mode $M2$ such that tasks T5, T6 and T7 will be removed from Core 1 and Core 2 (i.e. T8 and T9 are *finished tasks*). Similarly, assume that another mode change request (MCR2) is triggered at another engine speed and initiates the transition from the operational mode $M2$ to the operational mode $M3$ such that tasks T5, T6 and T7 are removed from Core 1 and Core 2 (i.e. T5, T6 and T7 are *finished tasks*). Corresponding to these two mode changes one can modell the opposite transitions from mode $M3$ to $M2$ where tasks T5, T6 and T7 are (re-)added on the cores and from $M2$ to $M1$ where tasks T8 and T9 are (re-)added on the cores. T1 and T4, the higher priority tasks on each core, represent *unchanged tasks* and execute independent of the mode changes.

To control the transition between operational modes a system designer can opt for *synchronous* or *asynchronous* mode change protocols, both types beeing supported by the AUTOSAR specifications related to the mode-management topic [10]. Asynchronous mode change protocols are characterized by higher responsiveness (i.e. allow new mode tasks start as early as possible after a mode change request). Therefore, they are most likely to be implemented in automotive systems where usually new mode actions must be performed as soon as possible. However, as discussed in Section 4.4.1 the overlapping execution of the different tasks under asynchronous mode change protocols leads to an increased load on the processor cores, which direclty translates into an increase of the tasks worst-case response times and potentially to deadline misses. To avoid harming the timing behavior of the multi-mode system considered in this use case, in what follows we discuss new options for the design and analysis of such multi-mode real-time systems.

4.4.4.3.2 Design Options for Applications with Engine-Synchronous Tasks

In the considered mode change example above, task modes are selected based on engine speed. This implies that at a threshold speed (e.g. a speed where the system switches from a high quality to a low quality mode), the system can be in one of two modes, depending on whether the vehicle is accelerating or decelerating. Both situations need to be considered in order to identify the most critical scenario, and to choose threshold values correspondingly. With respect to our case study, the question is:

What are the engine speeds at which mode changes have to be initiated such that (i) the impact on the systems timing is minimum and (ii) the timing constraints are certainly met on all cores?

This question was relatively easy to answer for single-core setups. But, mode changes for distributed and multi-core systems imply a more complex behavior where the load

change during execution is not necessarily monotonous and propagates between the communicating tasks on the different cores (see Section 4.4.1). Thus, an analysis is required that allows to quantify the mode change latency and the peak load and task response times during all transitions (illustration in Figure 4.13). Based on these values,

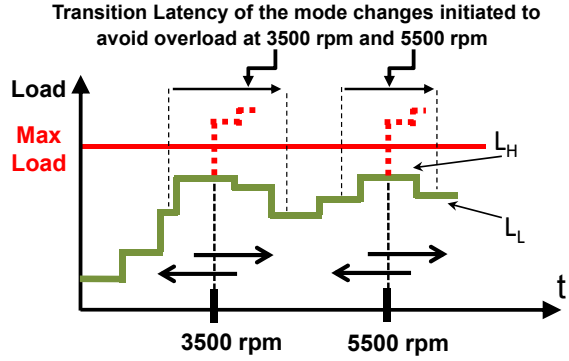


Figure 4.13: Multiple mode changes in order to avoid overload at different RPM values.

the threshold rpm values can be computed:

i When *accelerating*, the mode change has to be initiated (i.e. trigger a mode change request MCR) in sufficient time before the engine speed imposes a non-schedulable situation at a critical point CP rpm. For example, assume task T8 and T9 in Figure 4.10 have to be dropped-off at 3500rpm (and similarly T5, T6 and T7 at 5500rpm) in order to avoid an overload situation e.g. at the critical point CP1=3600rpm (and CP2=5600rpm, respectively). Instead of just dropping-off these tasks a controlled removal of them should be initiated in enough time before the system reaches a non-schedulable situation.

   The mode change transition latencies (see Section 4.4.1 and 4.4.2), corresponding to the mode changes that consist of stopping tasks, have to be computed for different assumptions regarding the moment of triggering the MCR. The calculation can be performed with the analysis method introduced in Section 4.4.2.

   For each critical point CP rpm, the engine speed X rpm (X < CP) will be identified such that the duration of the mode change during acceleration initiated at X rpm (i.e. the mode change transition latency for acceleration at X rpm denoted here with LA(X)) is less than the time the engine needs to accelerate from X rpm to CP rpm (i.e. Δt(X,CP) - see system model in Section 4.4.4.1).

ii When *decelerating*, the mode change that aims at restarting tasks can only be initiated when the engine speed indicates sufficient headroom in order to allow successful scheduling also during the mode change transition. When decelerating from 5500rpm to 3500rpm a change from a low level load to a high level load is performed. The previously dropped tasks could be restarted too early to each other, fact that would lead to an overlap of multiple mode changes. As indicated in Section 4.4.4.2 this could lead to an overload situation. Thus, the mode change transition latencies during

deceleration (denoted LD) have to be calculated for each required mode change.

Furthermore, when decelerating, a mode change that consists of restarting tasks should be initiated only if there is sufficient headroom in order to allow the successful scheduling also in case of a sudden acceleration, i.e. if there is enough time to restart the tasks during an acceleration from Y to X rpm. In other words, LD(Y) time units after the mode change request that triggers the restart of the tasks (i.e after the transition latency of the mode change initiated at Y rpm), the system has to reach a steady operational mode with stable tasks states such that these tasks can be safely removed again starting at X rpm. Thus, the engine speed Y rpm (Y < X) at which a mode change is allowed to be initiated during deceleration has to be identified such that the duration of the mode change during an acceleration initiated at Y rpm is less than the time the engine needs to suddenly accelerate from Y rpm to X rpm (i.e. LD(Y) = LA(Y) < Δt(Y,X)).

For each critical point CP rpm the timing constraints will not be harmed if the system is designed such that mode change transition latencies exhibit a hysteresis around an engine speed X rpm (X < CP) that can be identified as indicated above at (i) and (ii). An example is illustrated in Figure 4.14a and 4.14b. Due to mechanical characteristics

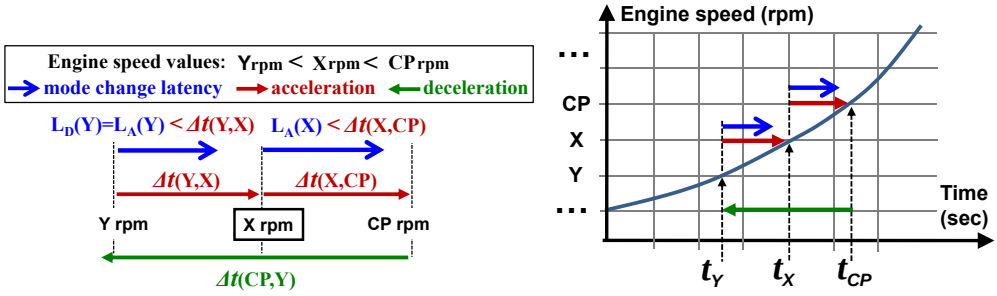


Figure 4.14: a) Mode changes shall be initiated at X rpm during acceleration and at Y rpm during deceleration in order to avoid a non-schedulable situation at CP rpm. b) Complex mode changes are possible if there is enough headroom for mode change transition latencies.

(e.g. flywheel inertia) the time an engine needs to accelerate or decelerate is large (see Figures 4.11b and 4.14b) in comparison to the execution time of the functions on the engine control units. Thus, if fast mode changes can be guaranteed, mode change protocols can be employed in order to avoid the service degradation resulting from the overly pessimistic measures applied in the current practice. The analysis solution proposed in Section 4.4.2 allows taking into account the mode transition latencies and thus enables the safe provisioning of multi-mode distributed applications in single-core and multi-core environments.

4.5 Response-Time Analysis for Multi-Mode Applications on Multi-Core Systems with Shared Resources

The previous section addressed the timing behavior of multi-mode distributed application without taking into account the timing dependencies caused by the common use of shared resources. As already discussed in the introductory part of this chapter, the problem of sharing resources by multi-mode applications was studied before only in the context of single-core processor systems. This section introduces an approach for safely handling inter-core and intra-core shared resources across asynchronous mode changes in multi-core setups and provides a blocking- and response-time analysis method that suits the next generation AUTOSAR conform multi-core processors.

4.5.1 Multi-Mode Multi-Core System Model

The multi-mode multi-core system model we introduce next combines elements of the multi-core system model in Section 3.3 with the multi-mode system model in Section 4.3. More exactly, for the purpose of this section we consider all elements of the multi-mode system model in Section 4.3 and add the following extension: the different types of arbitrarily activated multi-mode tasks (i.e. added, finished and unchanged) in the set $\mathcal{T} = \{\tau_1, \ldots \tau_n\}$ are assumed to be statically mapped on a multi-core architecture which consists of:

(i) a set of m processor cores ($m \geq 2$), each core being individually scheduled by a static priority preemptive (SPP) scheduler;

(ii) local shared resources (LRs), which are restricted to individual cores, and global shared resources (GRs), which can be accessed from each of the m cores.

Shared resources are assumed to be objects that require serialized access. For their arbitration we consider: for local shared resources the PCP [116, 100] and for global shared resources the AUTOSAR spinlock-based shared resource arbitration mechanism [12]. During execution each job of a task can perform multiple non-nested accesses to local shared resources and global shared resources. Each task access to one of these shared resources is considered a critical section guarded by a semaphore and protecting a local or a global resource. We differentiate between local critical sections (lcs) and global critical sections (gcs). The size of a lcs or of a gcs when it is accessed by jobs of a task τ_i are denoted ω_i^{LR} or ω_i^{GR}. With $\tilde{\eta}_i^{GRx}$ or $\tilde{\eta}_i^{LRx}$ we denote the load imposed by a job J_i on a global resource GRx or a local resource LRx.

An example of a multi-mode multi-core system during a transition phase between two modes is illustrated in Figure 4.15. We assume that a MCR imposes a mode change that consists in removing task τ_{1F} from Core 1 and adding tasks τ_{3A}, τ_{5A} on Core 1, and τ_{6A} on Core 2. The unchanged tasks τ_{2U} and τ_{4U} execute independent of the mode change. I_1 to I_5 represent the event sources (given by the functions η^+ and δ^- - see Figure 2.1) at the tasks input. The local and the global resources (i.e. $LR1$, $LR2$ and $GR1$, $GR2$) are accessed as indicated with the dashed lines.

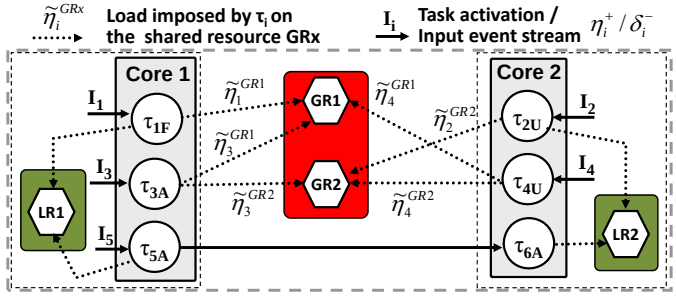


Figure 4.15: Multi-mode multi-core system during a transition phase.

4.5.2 Handling Shared Resources in Multi-Mode Multi-Core Systems using AUTOSAR 4.0

In order to handle the multi-mode behavior of an AUTOSAR 4.0 conform partitioned multi-core system, the execution of the different types of tasks (τ_F, τ_A, and τ_U) has to be considered when dealing with the arbitration of accesses to local (LR) and global (GR) shared resources.

For the arbitration of local resources the AUTOSAR OS uses on individual cores the priority ceiling protocol PCP inherited from the single-core OSEK OS [100]. According to OSEK-PCP, each semaphore associated to a LR is allocated offline a static priority ceiling which is equal to the highest priority of all task which access that LR. At runtime, when a task locks the semaphore corresponding to LR it immediately inherits its associated priority ceiling. In literature this implementation version of the Priority Ceiling Protocol is known as Immediate Priority Ceiling Protocol (IPCP) [118]. Remember that from the scheduling point of view the worst-case behavior of the IPCP and of the classic PCP (described in Chapter 2.4 in [116]) is identical.

In multi-mode systems, the obvious procedure for handling LRs is to allocate LRs multiple priority ceilings, one for each mode [153, 118]. Whereas this procedure is valid for individual modes, it can't be used during the transition phases controlled by asynchronous mode change protocols because [153, 118]:

(i) if priority ceilings have to be raised but are adjusted too late, then an added task, released after the MCR, may inherit an old mode priority ceiling which is lower than its current priority. This violates the IPCP, as priority ceilings must never be lower than the priority of any task using the resource;

(ii) if priority ceilings have to be lowered but are adjusted too early then a finished task may inherit a new mode lower priority ceiling. Thus, activations of the finished tasks, executed after the MCR, could experience increased blocking in comparison to the activations executed before the MCR.

Both situations invalidate the existing blocking time analysis methods and counter the timing behavior of real-time systems. In order to avoid the violation of the IPCP, for each LR_k ($k \in \mathbb{N}$) a unique ceiling priority $CP(LR_k)$ has to be assigned to be valid for all operating modes in the set M.

Theorem 4.7 ***(Ceiling of ceilings priority [139, 153, 118])*** *For each local resource LR_k, the only priority ceiling that is valid for all operating modes and all transitions between them is the so-called "ceiling of ceilings" priority, that corresponds to the highest priority [8] of any task τ_i accessing it in any mode $M_z \in M$ $(z \in \mathbb{N})$:*

$$\forall LR_k\ (k \in \mathbb{N}),\ \forall M_z \in M\ (z \in \mathbb{N}),\ \forall \Phi_{M_y}^{M_z} \in \Phi,\ \forall \tau_i \in T \ \text{ and } \tau_i \text{ uses } LR_k :$$

$$CP(LR_k) = \min(i) \tag{4.11}$$

Proof: By assuming all multi-mode tasks on each core as unchanged, one gets a single-mode worst-case system configuration where all tasks are simultaneously considered for scheduling. In such a setup ceiling priorities for shared resources can be safely fixed according to the OSEK-PCP. Even if at runtime some tasks will be removed or added as a consequence of the mode change requests, the shared resource ceiling priorities remain unchanged at the highest possible priority level. □

Regarding the arbitration of global shared resources specifications of the AUTOSAR 4.0 define the following [12]: during execution, a task τ_i will actively wait (spin) if a requested GR is occupied by a remote task; during active waiting a task may be preempted by higher priority local tasks, but lower priority local tasks cannot start executing; if a task locks a GR it suspends all interrupts on his host core and thus it becomes non-preemptable; nested accesses to GRs are not allowed; if nesting is required, an explicit partial ordering of calls for GRs has to be predefined offline in order to avoid deadlocks and potential starvation situations. As discussed in Section 3.9.2.1 AUTOSAR does not specify implementation details of the data structure associated to global semaphores. For the purpose of this thesis we assume that each global semaphore has a priority-ordered queue associated. In this way, when a task needs to lock a global resource and this is currently held by another task, the task queues itself on the semaphore queue. This means that in case of multiple coinciding requests for a certain global shared resource, the highest priority job requesting it will get the lock on the associated semaphore.

The key aspect of the AUTOSAR spinlock-based synchronization mechanisms is that global shared resources are arbitrated without using priority ceilings. The following corollaries follow:

Corollary 4.8 *Sharing global resources in AUTOSAR 4.0 conform multi-core systems does not require priorities to be dynamically adjusted when changing modes under asynchronous mode change protocols.*

Corollary 4.9 *In AUTOSAR 4.0 conform multi-mode multi-core systems, where accesses to global shared resources are arbitrated with the help of priority-based queues but without using priority ceilings, and where for the arbitration of accesses to local shared resources the "ceiling of ceilings" strategy is used, there is no danger of violating the resource arbitration policy and the statically assigned tasks' priorities. Implicitly blocking- and response-time analysis methods can be safely applied.*

[8] according to the system model this is indicated by the lowest task index.

In Section 4.5.3 we will introduce a timing analysis method for multi-mode applications scheduled on AUTOSAR 4.0 conform multi-core systems. Demonstrating that the blocking- and the response-times are bounded under all circumstances, we implicitly show that the procedure above for handling local and global shared resources across asynchronous mode changes is safe.

Before that, consider the scheduling examples in Figure 4.16 for the system in Figure 4.15. In the scheduling example in Figure 4.16a), during the transition phase initiated

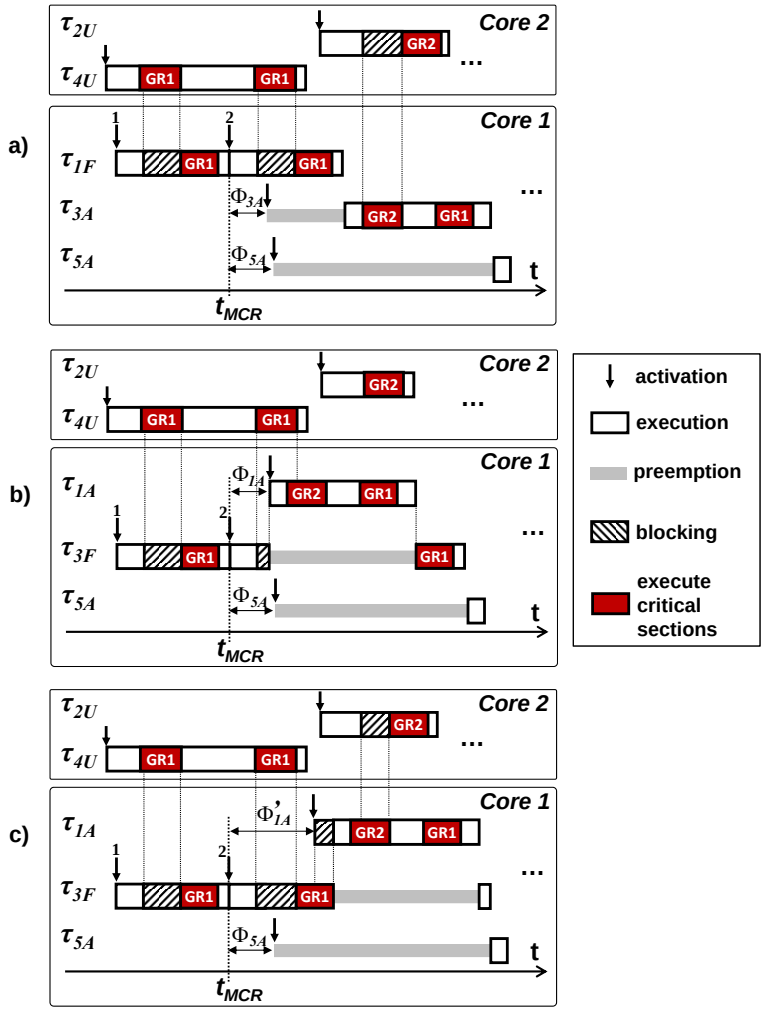


Figure 4.16: Scheduling examples for the dual-core system in Figure 4.15 when a) task priorities correspond to the system model in Figure 4.15; b) task τ_{3A} has higher priority than τ_{1F}, i.e. their priorities are interchanged, and the offset of the added task remains unchanged, i.e. $\Phi_{3A} = \Phi_{1A}$; and c) priorities of tasks τ_{3A} and τ_{1F} are interchanged and the offset of the added task is larger in comparison to case b), i.e. $\Phi'_{1A} > \Phi_{1A}$.

at t_{MCR}, tasks τ_{3A} and τ_{5A} cannot start executing before task τ_{1F} finishes all activations corresponding to the old mode, i.e. initiated not later than t_{MCR}. According to the AUTOSAR specification, as τ_{1F} has higher priority than τ_{3A} and τ_{5A} even if τ_{1F} is blocked by the remote task τ_{4U}, the tasks τ_{3A} and τ_{5A} will not execute. Thus, lower priority local new mode (added) tasks cannot influence the execution of old mode (finished) higher priority local tasks through the usage of global shared resources.

However, if the priorities of task τ_{3A} and τ_{1F} would be interchanged then τ_{3A}'s execution might be delayed by a lower priority local finished or unchanged task. As illustrated in Figure 4.16b) and c) the occurrence of such a blocking scenario depends on the offset the added task is released after the MCR.

As shown in Figure 4.16b), if the added task is released during the normal execution or during the busy-waiting of a lower priority task, the AUTOSAR scheduling and resource arbitration allow the added task preempt the task holding the processor and start executing without further local blocking times. However, as shown in Figure 4.16c), if the offset is large and a lower priority local task (in our example a lower priority finished task) manage to lock a global resource, it disables all interrupts and therewith blocks the added task. This blocking time is however limited to the time one lower priority local task holds the lock on a shared resource.

The blocking scenarios across asynchronous mode changes in systems with more than two cores are not much different. Assuming that an added task with the lowest priority in the system would be mapped on a third core (see task τ_{7A} in the scheduling scenario in Figure 4.17) and would start executing before τ_{1F} finishes, this could queue up for the global resource $GR1$ and even lock it. For τ_{1F}, the highest priority task in the system, this blocking scenario is not different to the one depicted in Figure 4.16a) where the lower priority remote tasks τ_{4U} can block it during the transition phase. As tasks are

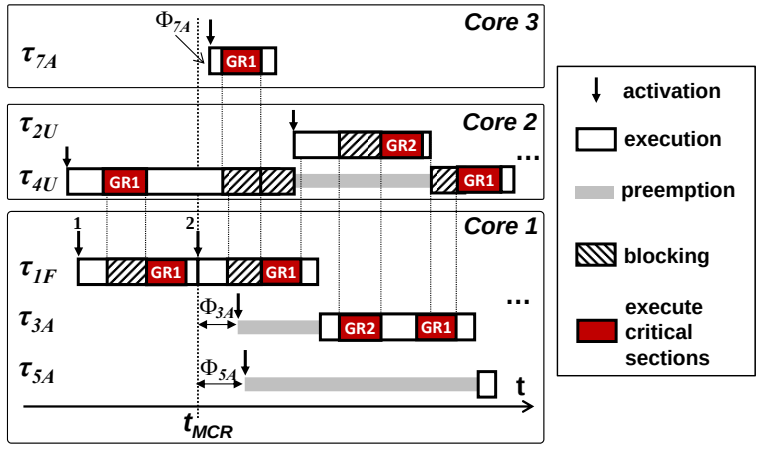


Figure 4.17: Scheduling example for the case in the system in Figure 4.15 there would be a third core on which a lower priority added task τ_{7A} would be started during the transition phase.

queueing themselves on the priority-based queue associated to each global semaphore, task τ_{1F} can be blocked by only one lower priority remote task. Thus, the blocking time of task τ_{1F}, corresponding to this blocking scenario, is in general equal to the size of either τ_{7A}'s or of τ_{4U}'s global critical section.

A significant difference can be observed in this particular case for the unchanged task τ_{4U} on Core 2. This task will be blocked once by the lower priority task τ_{7A} and then by the higher priority remote tasks τ_{1F} and τ_{3A} running on Core 1.

From the scheduling examples in Figure 4.16 and 4.17 we can see that in multi-mode multi-core systems higher priority new mode (added) tasks can be blocked by old mode (finished) tasks and higher priority old mode (finished) tasks can be blocked by lower priority new mode (added) tasks, however, only for short time intervals bounded by the size of critical sections. Blocking scenarios in which higher priority tasks block lower priority tasks are of course possible. But, this behavior does not violate the arbitration strategies and corresponds to the desired AUTOSAR functionality where more urgent tasks have to be executed first.

Inherently, the AUTOSAR spinlock-based arbitration strategy avoids the problems identified in case of "classically" using priority ceilings for local shared resources [153, 118] under asynchronous mode change protocols. Next, all possible blocking scenarios will be covered and upper bounded by the blocking time terms introduced in Section 4.5.3.2. These terms will be further integrated in the response-time analysis procedure.

4.5.3 Timing Analysis for Multi-Mode Multi-Core Systems with Shared Resources

In order to derive the blocking- and the response-time analysis for multi-mode multi-core systems with shared resources, we rely on concepts from the real-time multiprocessor and multi-mode scheduling theory. More exactly, we rely on the *classic busy window* technique in [154] which was already used and extended in order to analyse the timing behavior of (i) multi-mode systems [118, 65, 89] (see Section 4.4) and (ii) multi-core systems with shared resources [130] (see also Section 3.7.2, 3.8.2 and 3.9.3).

As known from the previous chapters, the level-i busy window of a task τ_i is generally defined as the time interval for which a resource executes only tasks of priority greater than or equal to the priority of task τ_i and during which the resource is never idle [154]. The maximum level-i busy window of q activations of a task τ_i under preemptive scheduling in partitioned multi-core systems with shared resources and where tasks do not suspend [9] when waiting for shared resources can be obtained by iteratively solving the following equation [10]

[9] As discussed in Section 3.8.2.1 and 3.9.3.1, under AUTOSAR 4.0 arbitration tasks do not suspend when waiting for shared resources and therefore the critical instant scenario and the calculation of the maximum busy windows are not influenced by the effect of deferred execution identified in [116].

[10] Equation (4.12) is similar to (3.19) in Section 3.7.2.2 with the difference that the effect of deferred execution, in 3.19 captured by the response time term, does not have to be considered.

$$w_i^{n+1}(q) = q \cdot C_i + BT_i(w_i^n(q)) + \sum_{\forall \tau_j \in hpl(i)} \eta_j^+(w_i^n(q)) \cdot C_j \tag{4.12}$$

where $w_i^n(q)$ is the maximum busy window of q activations of task τ_i with $q = 1, \ldots Q_i$ and $Q_i = \min\{q \geq 1 | w_i(q) < \delta_i^-(q+1)\}$, i.e. the iteration has to be continued as long as new activations of τ_i arrive before the previous finish; $BT_i(w_i^n(q))$ is the maximum blocking time of τ_i in $w_i(q)$; $\eta_j^+(w_i(q)) \cdot C_j$ is the interference τ_i suffers due to the maximum workload of a higher priority job τ_j in $w_i(q)$.

The worst-case response time of a task τ_i is given by the largest response time of any of the q ($q = 1, \ldots Q_i$) task activations that lie within the maximum level-i busy window. The response time of the q-th activation of task τ_i is given by the difference between the window length $w_i(q)$ and the moment when this activation was initiated relative to the beginning of the busy interval. This is given by $\delta_i^-(q)$. The WCRT of any task τ_i is conservatively obtained with

$$R_i = \max_{q=1..Q_i} (w_i(q) - \delta_i^-(q)) \tag{4.13}$$

and the schedulability test consists in checking whether the condition $R_i \leq D_i$ holds for every task τ_i in the system.

The classic response-time analysis procedure, which uses equations (4.12) and (4.13) above, can be applied only for single-mode system configurations and implicitly to each individual operational mode. As discussed in Section 4.4.2.1 and as identified in literature [118, 65, 89] the worst-case response times of tasks during a transition phase can be obtained by deriving the *maximum transition busy window* (i.e. the maximum busy window during which a MCR occurs). In order to calculate maximum transition busy windows and therewith worst-case response times in multi-mode multi-core systems with shared resources, in what follows we extend equation (4.12) to consider the execution of different types of multi-mode tasks and introduce the blocking-time analysis procedure that corresponds to the term BT in (4.12).

4.5.3.1 Maximum Transition Busy Window in Multi-Core Systems

Our goal is to safely bound the timing behavior of tasks in multi-mode multi-core systems with shared resources. For that purpose, we compute the maximum transition busy window (abbr. MTBW) for each task by:

i) identifying the worst-case scenario when the MCR shall occur such that it certainly leads to the worst-case execution during the transition phase and

ii) determining the maximum workload (denoted MW) of the different types of multi-mode tasks (i.e. finished, added, and unchanged) and their maximum blocking time in case of sharing resources for the identified worst-case mode change scenario.

4.5.3.1.1 Worst-Case Mode Change Scenario in Multi-Core Systems

Two aspects must be jointly handled in order to determine the worst-case timing behavior of any task in a multi-mode system, namely the criteria for constructing the

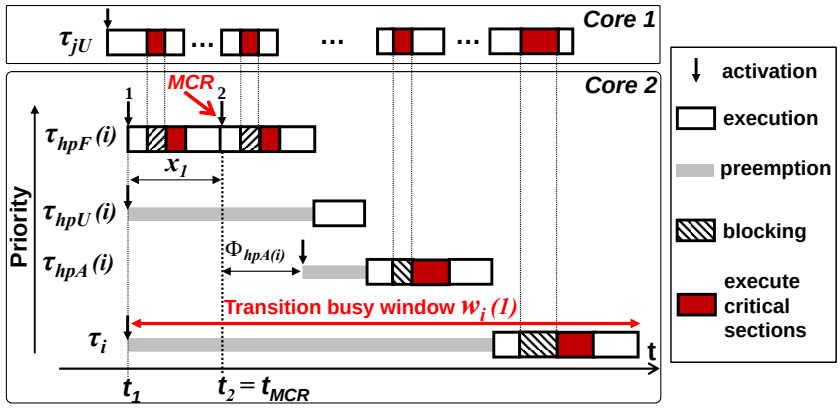


Figure 4.18: Scheduling example for a task τ_i during a mode change where MCR coincides with the 2nd activation of the higher priority finished task.

worst-case mode change scenarios and the procedure for identifying the worst-case mode change scenario among the constructed ones.

Criteria for the Construction of Worst-Case Mode Change Scenarios.

According to Theorem 4.1 (see also [118, 65]) the worst-case mode change scenario for a task τ_i is obtained when: **(1)** t_{MCR} coincides with the activation instant of a finished higher priority task in $hp_F(i)$; **(2)** added tasks ($hp_A(i)$) are released with an offset $\phi_{hpA(i)}$ after the initiation of the MCR and **(3)** unchanged higher priority local tasks in $hp_U(i)$ are assumed released simultaneously with τ_i, i.e in the classical critical instant.

These arguments, valid for uni-processor systems without shared resources, have to be investigated in the context of AUTOSAR conform multi-core setups. For explanations we refer to the scheduling example in Figure 4.18.

According to the AUTOSAR arbitration policy for global shared resources [12] (see also Section 4.5.2) a task which has an outstanding request for a shared resource will actively wait for that resource without suspending. This means that lower priority local tasks cannot start executing as long as a higher priority local is running or busy-waiting.

Such a scheduling example is illustrated in Figure 4.18 for task τ_i, which is assumed to be the analyzed task. As can be seen, even if each request of the tasks $\tau_{hpF}(i)$ and $\tau_{hpA}(i)$ on Core 2 for a global shared resource is blocked by the remote task τ_{jU} on Core 1, the lower priority tasks τ_i on Core 2 cannot start executing. This means that for task τ_i the busy-waiting times of higher priority local tasks represent an extension of their core execution times. This holds independent of the type of tasks.

Thus, in multi-mode multi-core systems using the AUTOSAR synchronization mechanism the three arguments above (i.e. **(1)**,**(2)** and **(3)**) for constructing worst-case mode change scenarios remain valid.

Identification of the Worst-Case Mode Change Scenario.

Regarding the identification of the worst-case mode change scenario, in case of arbitrary activated tasks there may be multiple higher priority finished tasks (tasks in

$hp_F(i)$) and for each of these tasks there may be several possible activations (i.e. jobs) released at different moments in time, e.g. t_1 and t_2 in Figure 4.18. As discussed in Section 4.4.2.1, in order to find the worst-case transition scenario one must identify all time instances where the occurrence of the MCR should be assumed. The moments in time corresponding to the activations of the hp_F tasks are relative to the occurrence of the MCR at t_{MCR}. Let X_i be the set of all possible time intervals x_i, computed with the Algorithm 4.1 on page 144 (see also [65]), relative to t_{MCR} which have to be investigated. The largest busy window obtained for one of the values x_i represents the *maximum busy window* of a task τ_i during which a MCR occurs.

The general procedure is the same as the one introduced in Section 4.4.2.1 for the analysis of single-core multi-mode systems. The key difference in case of multi-mode multi-core systems with shared resources is given by the inter-core timing dependency caused by the blocking times of tasks that execute on different processing cores. As during transition phases tasks of different types (i.e. added, finished and unchanged) can be simultaneously scheduled on different cores and can block each other, the blocking times have to be considered in order to identify the worst-case mode change scenario.

Essentially, this imply an adjustment of Algorithm 4.1 used for computing all possible values x_i. Algorithm 4.1 rely on the computation of the maximum busy window with a maximum workload of finished and unchanged tasks, i.e. the longest busy window within the old mode (see first line of Algorithm 4.1). For the analysis of multi-mode multi-core systems the terms for computing the maximum workload of finished and unchanged tasks have to be extended with a factor $BT_i(L_i)$ that captures the blocking times of these tasks during the investigated busy window L_i as follows:

$$\begin{aligned} L_i^{n+1} &= \sum_{\forall \tau_{jU} \in hpl_U(i)} (\eta^+_{\tau_{jU}}(L_i^n) \cdot C_{jU} + BT_{jU}(L_i^n)) \\ &\quad + \sum_{\forall \tau_{jF} \in hep_F(i)} (\eta^+_{\tau_{jF}}(L_i^n) \cdot C_{jF} + BT_{jF}(L_i^n)) \end{aligned} \tag{4.14}$$

The first sum term in (4.14) captures the maximum workload of higher priority unchanged tasks from the set $hpl_U(i)$ during L_i plus the blocking times of these tasks during L_i. As discussed earlier, for any task in a multi-core setup with an AUTOSAR conform spinlock-based shared resource arbitration mechanism the busy-waiting times (i.e. blocking times) of the higher priority local tasks represent and extension of their execution time. In order to maximize the busy window, the blocking times of these tasks have to be considered as workload. The second sum term in (4.14) captures the maximum workload and the blocking time of higher priority finished tasks from the set $hep_F(i)$ (i.e. incl. τ_i if this is a finished task) during L_i. Equation (4.14) can be solved by iteration if all its components (i.e. the maximum workload of the considered tasks and their blocking times) are order-preserving. This aspect will be proven in Section 4.5.3.4 for systems corresponding to the model introduced in Section 4.5.1 and 4.5.2. The calculation starts with an initial value $L_i(0) = 0$ and stops when two consecutive iterations provide identical values ($L_i^{n+1} = L_i^n$), or when some threshold (e.g. a real-time constraint) is exceeded.

Thus, by substituting the calculation of the maximum busy window L_i within the old mode scenario in Algorithm 4.1, for each task in a multi-mode multi-core system one can determine all time intervals x_i which have to be investigated in order to identify the worst-case mode change scenario and therewith the worst-case timing behavior.

4.5.3.1.2 Calculation of the Maximum Transition Busy Window (MTBW)

The maximum transition busy window (MTBW) for a task τ_i is obtained for one of the values x_i, which corresponds to time intervals relative to the occurrence of the MCR at t_{MCR}. In order to compute the MTBW for multi-mode multi-core systems, the busy window equation (4.12) for partitioned multi-core systems under static priority preemptive scheduling has to be extended to consider the maximum workload MW generated by the execution of unchanged, finished and added tasks. Additionally the maximum blocking time these tasks can experience when waiting for the requested shared resources have to be considered.

Thus, the maximum transition busy window in case of partitioned SPP scheduling of multi-mode multi-core systems with shared resources is obtained by iteratively solving (4.15):

$$\begin{aligned} w_i^{n+1}(q) = MW^i + BT_i(w_i^n(q)) + \sum_{\forall \tau_F \in hpl_F(i)} \eta^+_{\tau_F}(x_i) \cdot C_{\tau_F} + \\ \sum_{\forall \tau_U \in hpl_U(i)} \eta^+_{\tau_U}(w_i^n(q)) \cdot C_{\tau_U} + \\ \sum_{\forall \tau_A \in hpl_A(i)} \eta^+_{\tau_A}(w_i^n(q) - x_i - \phi_{\tau_A})_0 \cdot C_{\tau_A} \end{aligned} \tag{4.15}$$

with the maximum workload MW^i of the analyzed task τ_i:

$$MW^i = \begin{cases} q \cdot C_i; \text{ if } (i == U) \,||\, (i == F) \\ \min(q, \eta_i^+(w_i^n(q) - x_i - \phi_i)_0) \cdot C_i; \text{ if } (i == A) \end{cases} \tag{4.16}$$

- The first term in (4.15) is covered by the clauses of (4.16) which give the maximum workload MW^i of the analyzed task τ_i depending on its type as follows:
  - The first clause of (4.16) covers the case when the analyzed task τ_i is an unchanged (τ_{iU}) or a finished task (τ_{iF}). Even if the formula is identical, the difference between the calculation of the maximum workload for finished and unchanged tasks with $q \cdot C_i$ is given by the termination of the iterative calculation with (4.15).

    When analyzing an unchanged task τ_{iU} the iteration is performed for all jobs $q = 1, \ldots Q_i$ with $Q_i = \min\{q \geq 1 | w_i^n(q) < \delta_i^-(q+1)\}$. In other words, the iteration has to be continued as long as new activations of τ_{iU} arrive before the previous finish.

For a finished task τ_{iF} one has to iterate only over those jobs of τ_{iF} which are activated within x_i, i.e. only for those jobs which are activated before the occurrence of the MCR. This means the calculation is performed for all jobs $q = 1, \dots Q_i$ with $Q_i = \eta_i^+(x_i)$.

– The second clause of (4.16) covers the case when τ_i is an added tasks. This indicates that, for large values of the offset ϕ_i, task τ_i does not contribute to the busy window $w_i(q)$. The function $\eta^+_{\tau_A}(w_i^n(q) - x_i - \phi_{\tau_A})_0$ represents a modified version of the original upper event arrival function $\eta^+(\Delta t)$ and returns 0 if $w_i^n(q) - x_i - \phi_i < 0$.

- The second term in (4.15) captures the blocking time experienced by the analysed task due to the use of shared resources. Note that the blocking times also depend on the system's multi-mode behavior. Therefore, the factor $BT_i(w_i^n(q))$ in (4.15) has to be derived by considering the execution of the different types of tasks on all processing cores in the system. The blocking time analysis which upper bounds the term $BT_i(w_i^n(q))$ in (4.15) is subject of Section 4.5.3.2.
- The following three sum terms in (4.15) cover the MW due to the execution of higher priority finished, unchanged and added tasks. Activated but not completed jobs of the finished tasks are assumed occurring in a time interval x_i starting before the initiation of the MCR at t_{MCR}. Added tasks are considered released with an offset ϕ_{τ_A} after t_{MCR}.

Similar to (4.14) equation (4.15) can be solved by iteration if all its components grow with the window size (i.e. are order-preserving), aspect which will be addressed in Section 4.5.3.4 [11].

4.5.3.2 Blocking Time Analysis in Multi-Mode Multi-Core Systems

In this section, we introduce a blocking time analysis for arbitrarily activated tasks that share resources in an AUTOSAR conform multi-mode multi-core setup scheduled by a static priority preemptive (SPP) scheduler. Similar to the blocking time analysis equations presented across Chapter 3, the blocking time terms we introduce next capture the overlapping job executions during their busy windows w_i.

The parameters used in the blocking factors correspond to the system model in Section 4.5.1 and use the general terms listed in Table 3.1. These won't be repeated here, but remember that the *SharedResourceRequestBound* function and the sets of considered tasks have to capture the specific type (τ_U,τ_F,τ_A) of tasks that are subject of blocking.

Based on the procedure for handling shared resources in multi-mode multi-core systems using AUTOSAR, introduced in Section 4.5.2, the blocking time of a job J_i in a partitioned multi-mode multi-core system consists of the following factors:

[11] Equation 4.15 is similar to the busy windows equations (3.19), (3.28) and (3.45) proven as order-preserving in Section 3.10.

1. Local blocking time. Under AUTOSAR preemptive scheduling and OSEK-PCP [100] shared resource arbitration, a job J_i of a task τ_i can be blocked once by job J_j of a lower priority local task $\tau_j \in lpl(i)$. As nesting is not allowed, the lower priority local job J_j can either execute a local critical section *lcs* or a global critical section *gcs* of duration ω_j^{LR} or ω_j^{GR}. Of course, the lower priority local tasks that can block the analyzed task τ_i depend on their types. Thus, the local blocking time of a job J_i is bounded by the maximum length of a local or of a global critical section as follows:

$$LB_i(w_i(q)) = \max(\omega_j^{LR}, \omega_j^{GR}) \tag{4.17}$$

$$\text{with } \begin{cases} \tau_j \in lpl_U(i) \bigcup lpl_F(i); & \text{if } (i == F) \\ \tau_j \in lpl(i); & \text{if } (i == U) \,||\, (i == A) \end{cases}$$

The first clause above captures the case where τ_i is a finished task. In this case lower priority local tasks of type added (i.e. lpl_A) cannot start and queue up for any shared resource and thus these cannot block τ_i. The second clause captures the case where τ_i is an unchanged or an added task that can be blocked by one previously released job of a task τ_j of any type, i.e. $\tau_j \in lpl(i) = lpl_U(i) \bigcup lpl_F(i) \bigcup lpl_A(i)$.

2. Direct blocking time. Each task τ_i can be blocked when trying to access a global resource (GR) if this has already been locked by a remote task with lower or higher priority.

Thus, each time a job J_i of the analyzed task τ_i attempts to lock a GR, it may find that this is currently locked by one of the jobs J_j of the lower priority remote tasks in the set $\theta_{i,j}$, i.e. by those tasks that are mapped on remote cores and access the same global resources as τ_i. In a worst-case scenario, each request for a GR of a job J_i can be blocked for the duration of the longest global critical sections ω_j^{GR} of a lower priority remote task in the set $\theta_{i,j}$. This is captured by:

$$DB_{i,lpr}(w_i^n(q)) = q \cdot n_i^G \cdot \max_{\forall \tau_j \in \theta_{i,j}} (\omega_j^{GR}) \tag{4.18}$$

Of course, the remote tasks in $\theta_{i,j}$ can be of different types, i.e. added, finished and unchanged. From the worst-case perspective always considering the largest global critical section of any of the tasks in $\theta_{i,j}$, independent on its type, is safe [12].

In addition to jobs of the lower priority remote tasks, each job J_i can also be blocked by higher priority remote jobs that access the same GR as J_i (i.e. by jobs of tasks in the set $\Theta_{i,j}$). As opposed to lower priority remote jobs, higher priority remote jobs may be served multiple times before jobs of task τ_i will be able to lock the requested GRs. Therefore, the load $\tilde{\eta}_j^+$ imposed by higher priority remote tasks on the GRs accessed

[12] This assumption can be also pessimistic in case the largest *gcs* that can ever block task τ_i belong to an added tasks but this is released on the remote core with a large offset after the MCR. For an exact calculation, the implementation of the blocking time term $DB_{i,lpr}$ has to consider different lengths of *gcs* depending on the tasks' execution during the transition busy window $w_i^n(q)$ of task τ_i.

by τ_i during the transition busy window $w_i^n(q)$ has to captures the type of the blocking task, as follows:

$$\forall \tau_j \in \Theta_{i,j} :$$

$$\tilde{\eta}_j^+ = \begin{cases} \tilde{\eta}_j^+(x_j); \text{ if } \tau_j \in hpr_F(i) \bigcap \Theta_{i,j} \\ \tilde{\eta}_j^+(w_i^n(q)); \text{ if } \tau_j \in hpr_U(i) \bigcap \Theta_{i,j} \\ \tilde{\eta}_j^+(w_i^n(q) - x_i - \phi_j); \text{ if } \tau_j \in hpr_A(i) \bigcap \Theta_{i,j} \end{cases} \tag{4.19}$$

Thus, the direct blocking time due to higher priority remote tasks is given by:

$$DB_{i,hpr}(w_i^n(q)) = \sum_{\forall \tau_j \in \Theta_{i,j}} \tilde{\eta}_j^+ \cdot \omega_j^{GR} \tag{4.20}$$

with $\tilde{\eta}_j^+$ given by (4.19).

As can be observed in equation (4.19) and (4.20) the blocking time of a task τ_i, investigated for one time interval x_i relative to t_{MCR}, depends on the time intervals x_j relative to t_{MCR} that have to be investigated for tasks τ_j on other cores. This dependency can be handled by integrating the blocking-time analysis into a compositional system-level analysis procedure [64, 32] as discussed in Section 2.3 and 3.10.

The worst-case direct blocking time $DB_i(w_i^n(q))$ a task τ_i can encounter in a time window $w_i(q)$, when executing on a multi-mode multi-core system is given by the sum of the two blocking factors in (4.18) and (4.20):

$$DB_i(w_i^n(q)) = DB_{i,lpr}(w_i^n(q)) + DB_{i,hpr}(w_i^n(q)) \tag{4.21}$$

3. Indirect blocking time / Busy-waiting of higher priority local tasks. According to the AUTOSAR specification tasks do not suspend when waiting for the requested GR but keep spinning until the resource becomes available or a higher priority local task preempts it. Thus, a job J_i cannot start executing on its host core as long as higher priority local tasks are actively waiting for the required GRs, which means that the direct blocking times of the higher priority local tasks prolong the delay of task τ_i. In other words, the indirect blocking time of a task τ_i is given by the direct blocking times of the higher priority local tasks that can preempt the analyzed task τ_i.

As already known from the direct blocking scenario considered above, a task can be blocked several times by multiple remote tasks. This holds not only for the analyzed task τ_i but also for the higher priority local tasks which can preempt τ_i (i.e. $\tau_k \in hpl(i)$) during its execution outside critical sections or during busy-waiting. Similar to τ_i, requests for global resources of each job J_k of higher priority local tasks $\tau_k \in hpl(i)$ can be directly blocked by remote tasks with lower or higher priority, i.e. by tasks $\tau_j \in \theta_{k,j} \bigcup \Theta_{k,j}$. Thus, the indirect blocking time a task τ_i will experience in a multi-core setup due to the direct blocking of the higher priority local tasks $\tau_k \in hpl(i)$ can be derived with an equation similar to (4.21) as follows:

$$
\begin{aligned}
IB_i(w_i^n(q)) &= \sum_{\forall \tau_k \in hpl(i)} DB_k(w_i^n(q)) \\
&= \sum_{\forall \tau_k \in hpl(i)} [DB_{k,lpr}(w_i^n(q)) + DB_{k,hpr}(w_i^n(q))] \\
&= \sum_{\forall \tau_k \in hpl(i)} [\eta_k^+(w_i^n(q)) \cdot n_k^G \cdot \max_{\forall \tau_j \in \theta_{k,j}} (\omega_j^{GR}) + \sum_{\forall \tau_j \in \Theta_{k,j}} (\tilde{\eta}_j^+(w_i^n(q)) \cdot \omega_j^{GR})]
\end{aligned}
$$

However, the tasks that can preempt the analyzed task τ_i can be of type finished, added or unchanged. Therefore, the calculation of the maximum number of activations of the tasks in the set $hpl(i)$ within the transition busy window of task τ_i has to capture the multi-mode behavior of the different task types. Thus, the indirect blocking time equation above can be rewritten as [13]:

$$
\begin{aligned}
&IB_i(w_i^n(q)) = \qquad (4.22)\\
&\sum_{\forall \tau_{kA} \in hpl_A(i)} [\eta_{\tau_{kA}}^+(w_i^n(q) - x_i - \phi_{\tau_{jA}})_0 \cdot n_{kA}^G \cdot \max_{\forall \tau_j \in \theta_{kA,j}} (\omega_j^{GR}) + \sum_{\forall \tau_j \in \Theta_{kA,j}} (\tilde{\eta}_j^+ \cdot \omega_j^{GR})] + \\
&\sum_{\forall \tau_{kU} \in hpl_U(i)} [\eta_{\tau_{kU}}^+(w_i^n(q)) \cdot n_{kU}^G \cdot \max_{\forall \tau_j \in \theta_{kU,j}} (\omega_j^{GR}) + \sum_{\forall \tau_j \in \Theta_{kU,j}} (\tilde{\eta}_j^+ \cdot \omega_j^{GR})] + \\
&\sum_{\forall \tau_{kF} \in hpl_F(i)} [\eta_{\tau_{kF}}^+(x_i) \cdot n_{kF}^G \cdot \max_{\forall \tau_j \in \theta_{kF,j}} (\omega_j^{GR}) + \sum_{\forall \tau_j \in \Theta_{kF,j}} (\tilde{\eta}_j^+ \cdot \omega_j^{GR})]
\end{aligned}
$$

with $\tilde{\eta}_j^+$ given by:

$\forall Y \in \{A, F, U\}$ and $\forall \tau_j \in \Theta_{kY,j}$:

$$
\tilde{\eta}_j^+ = \begin{cases} \tilde{\eta}_j^+(x_j); \text{ if } \tau_j \in hpr_F(kY) \bigcap \Theta_{i,j} \\ \tilde{\eta}_j^+(w_i^n(q)); \text{ if } \tau_j \in hpr_U(kY) \bigcap \Theta_{i,j} \\ \tilde{\eta}_j^+(w_i^n(q) - x_i - \phi_j); \text{ if } \tau_j \in hpr_A(kY) \bigcap \Theta_{i,j} \end{cases} \qquad (4.23)
$$

The three clauses in (4.22) captures the influence of added, unchanged and finished tasks to the indirect blocking of the analyzed task τ_i.

In the first clause, the function $\eta_{\tau_A}^+(w_i(q) - x_i - \phi_{\tau_A})_0$, which indicates the maximum number of higher priority added tasks that can interfere with the execution of the analyzed task τ_i, represents a modified version of the original upper event arrival function $\eta^+(\Delta t)$ and returns 0 if $w_i(q) - x_i - \phi_i < 0$. Thus, higher priority added tasks can execute and initiate requests for a global resource only after the MCR occurence and after an release offset $\phi_{\tau_{jA}}$, more exactly not before $x_i + \phi_{\tau_{jA}}$ time units after the start of the transition busy window. Each of the n_{kA}^G accesses of a higher priority added task τ_{kA} to the global resources can be blocked by the largest critical section of a lower priority remote task τ_j in the set $\theta_{kA,j}$. Higher priority remote tasks in the set $\Theta_{kA,j}$ can be

[13] The literal index A, F and U associated to the task index k indicates explicitly the type of task.

of different types, added, unchanged and finished and can block the task τ_{kA} multiple times. This is captured by the right hand side term in the first clause in (4.22) which uses the clauses in (4.23).

The second and the third clause in (4.22) are similar to the first one and capture the execution and the blocking time of unchanged and finished tasks which have higher priority than τ_i.

4. Blocking when re-initiating cancelled requests for global resources. Each time a job J_i of the analyzed task τ_i is preempted while busy-waiting, its request for the global resource is cancelled. At the moment when J_i is re-scheduled and re-initiates the request for the global resource, it may be blocked by a remote job that could acquire the lock while J_i was preempted. Two aspects have to be considered in order to find an upper bound for this blocking type, namely (i) the maximum number of requests a task τ_i can re-initiate and (ii) the maximum time each of the re-initiated requests can be blocked:

(i) Regarding the maximum number of re-initiated requests of a task τ_i this is given by the maximum number of preemptions this task can experience during its transition busy window. As higher priority local tasks can be of different types, the maximum number of preemptions of τ_i depends on the maximum number of activations of the higher priority added, unchanged and finished tasks during the transition busy window as follows:

$$\sum_{\forall \tau_k \in hpl(i)} \eta_k^+(w_i^n(q)) = \sum_{\forall \tau_{kA} \in hpl_A(i)} \eta_{\tau_{kA}}^+(w_i^n(q) - x_i - \phi_{\tau_{jA}})_0 + \sum_{\forall \tau_{kU} \in hpl_U(i)} \eta_{\tau_{kU}}^+(w_i^n(q)) + \sum_{\forall \tau_{kF} \in hpl_F(i)} \eta_{\tau_{kF}}^+(x_i) \tag{4.24}$$

(ii) Regarding the maximum time each of the re-initiated requests can be blocked one has to identify the tasks that cause this blocking. In general, requests for global shared resources can be blocked once by one global critical section of a lower priority remote task and multiple times by global critical sections of higher priority remote tasks.

In a worst-case scheduling scenario, each re-initiated request of task τ_i or of the tasks that can preempt τ_i (i.e. $\tau_k \in hpl(i)$) can be blocked once by a lower priority remote task in the sets $\theta_{i,j}$ or $\theta_{k,j}$ [14]. for the duration of the longest global critical section $\max_{\forall \tau_j \in \theta_{i,j} \bigcup \theta_{k,j}} \left(\omega_j^{GR}\right)$. Of course, the remote tasks in $\theta_{i,j}$ can be of different types, i.e. added, finished and unchanged. However, from the worst-case perspective always considering the largest global critical section of any of the lowest priority remote tasks, independent on its type, is safe.

The influence of the higher priority remote tasks on task τ_i and on its higher priority local tasks $\tau_k \in hpl(i)$ is safely upper bounded in the direct blocking time and in the

[14] For an exact calculation, the highest priority task that can preempt τ_i has to be excluded from the set $\theta_{k,j}$. This is because the highest priority task in $\Psi(i)$ can preempt the execution of τ_i but its requests won't be re-initiated and thus not additionally blocked by a lower priority remote task.

indirect blocking time (i.e. in the direct blocking time of the higher priority local tasks) independent on the number of re-initiated requests.

Thus, the maximum possible blocking time of a task τ_i that results from τ_i or its higher priority local tasks being preempted while busy-waiting is captured by

$$CRB_i(w_i^n(q)) = \sum_{\forall \tau_k \in hpl(i)} \eta_k^+(w_i^n(q)) \cdot \max_{\forall \tau_j \in \theta_{i,j} \bigcup \theta_{k,j}} \left(\omega_j^{GR}\right) \tag{4.25}$$

with $\sum_{\forall \tau_k \in hpl(i)} \eta_k^+(w_i(q))$ given in this case by (4.24) above.

Overall Blocking Time. The worst-case blocking time $BT_i(w_i^n(q))$, as part of the maximum transition busy window computation with (4.15), that a task τ_i can encounter in a time window $w_i^n(q)$ is given by the sum of the four blocking factors above, i.e. (4.17), (4.21), (4.22) and (4.25)

$$BT_i(w_i^n(q)) = LB_i(w_i^n(q)) + DB_i(w_i^n(q)) + IB_i(w_i^n(q)) + CRB_i(w_i^n(q)) \tag{4.26}$$

4.5.3.3 Derivation of the Worst-Case Response Times

In order to derive the worst-case response times of tasks in multi-mode partitioned multi-core systems under static-priority preemptive scheduling, AUTOSAR conform shared resource arbitration and asynchronous mode change protocols the blocking times obtained with the equations in Section 4.5.3.2 are integrated in the maximum transition busy window computation with (4.15). Finally, the WCRT of a task τ_i is given by the largest response time R_i of any of the q activations ($q = 1..Q_i$, $Q_i = \min\{q \geq 1 | w_i(q) < \delta_i^-(q+1)\}$) that lie within the MTBW $w_i(q)$, i.e.

$$R_i = \begin{cases} \max(w_i(q) - \delta_i^-(q)); & \text{if } (i == U) \text{ || } (i == F) \\ \max(0, w_i(q) - x_i - \phi_i - \delta_i^-(q)); & \text{if } (i == A) \end{cases} \tag{4.27}$$

The clauses in (4.27) state that depending on the task's type, the response time R_i is obtained by subtracting from $w_i(q)$ the distance between the start of the transition busy window and the activation instant of the q-th job. If τ_i is an added task which is not activated within the transition busy window, R_i is 0. If worst-case response time values R_i are obtained for all the tasks in the multi-core system, the schedulability test consists of checking whether the condition $R_i \leq D_i$ holds for every task τ_i.

However, the response-time values can not be trivially calculated. As can be observed from (4.15), (4.19), (4.20), (4.22) and (4.23) the maximum transition busy window w_i and therewith the response time R_i of a task depend on the load $\tilde{\eta}_j^+$ imposed on the shared resources by tasks on other cores and potentially by their worst-case time interval x_j where the MCR shall occur. To solve this dependency the response- and the blocking-time analysis for multi-mode multi-core systems have to be integrated in the system-level compositional analysis procedure introduced in Section 2.3, similar to the system-level analysis integration for multi-core systems presented in Section 3.10.

4.5.3.4 System-Level Analysis Integration

The system-level analysis for AUTOSAR 4.0 conform multi-mode multi-core systems is an iterative analysis process which performs for each task on each core

1. the calculation of all possible time intervals x_i relative to the occurrence of the MCR which have to be investigated in order to derive the worst-case behavior during the transition phase (Section 4.5.3.1.1), i.e. to enable the calculation of the maximum transition busy windows (Section 4.5.3.1.2) and therewith of the worst-case response times (Section 4.5.3.3);
2. the calculation for each value x_i of the response-times with (4.27) (Section 4.5.3.3) which includes the computation of the maximum transition busy windows with (4.15) (Section 4.5.3.1.2); and
3. the calculation of the blocking times with (4.26) (Section 4.5.3.2) which requires the investigation of all possible time intervals x_j of other tasks on other cores.

until definite event models have been found (see Section 2.3 and 3.10). In case of a system-level convergence, the schedulability tests (i.e. test if $R_i \leq D_i$) have to be applied for each task in the system.

The iterative system-level analysis procedure represents a fixed-point problem, which can be solved only if the conditions of Corollary 2.2 are fulfilled for each local analysis procedure and each analysis parameter. The conditions demand that the analysis functions are order preserving with respect to their input parameters and that the set of the analysis results forms a complete partial order.

Order Preservation on Complete Partially Ordered Sets.

The building blocks of the system-level analysis procedure are the local response-time analyses based on the busy window approach [154]. Thus, the response-time and the busy window analysis functions for static-priority preemptive scheduling under asynchronous mode change protocols, as considered in this chapter, represent the central elements of the system-level approach and must adhere to the conditions of Corollary 2.2.

Theorem 4.10 *The response-time analysis and the busy window analysis of tasks in multi-mode multi-core systems under partitioned multiprocessor static-priority preemptive scheduling, AUTOSAR 4.0 shared resource arbitration and asynchronous mode change protocols are order preserving.*

Proof: We have to show that for each analysis state achieved by iteration the response-time analysis delivers increasing response time values. More exactly, we have to show that for two successive parametrizations j and $j+1$ of the event model EM_i associated to task τ_i (see Definition 2.7 and (2.9) and (2.10)), i.e. for the event model estimate EM_i^j of task τ_i in the analysis state as_j and the event model estimate EM_i^{j+1} of task τ_i in a successive analysis state as_{j+1} we have:

$$EM_i^j \leq EM_i^{j+1} \Rightarrow R_i(EM_i^j) \leq R_i(EM_i^{j+1})$$

Step 1. The response time R_i, calculated with (4.27) which is

$$R_i = \begin{cases} \max(w_i(q) - \delta_i^-(q)); & \text{if } (i == U) \mid\mid (i == F) \\ \max(0, w_i(q) - x_i - \phi_i - \delta_i^-(q)); & \text{if } (i == A) \end{cases} \quad (1)$$

is order preserving if all its elements, i.e. $\delta_i^-(q)$, x_i, ϕ_i and the maximum transition busy window $w_i(q)$, are order preserving with respect to the analysis states.

i. $\delta_i^-(q)$ - The event model estimates EM_i, given by the functions $\eta^+(\Delta t)$ and $\delta^-(n)$, have been proven to form an complete partial ordered set (see Chapter 3 in [142]):

$$EM_i^j \leq EM_i^{j+1} \Rightarrow \\ \forall q : \delta_i^{j,-}(q) \geq \delta_i^{j+1,-}(q) \Rightarrow \forall\ \Delta t \geq 0 : \eta_i^{j,+}(\Delta t) \leq \eta_i^{+,j+1}(\Delta t)$$

This means that whereas the minimum distance $\delta_i^-(q)$ between any q task activations may only decrease or remain unchanged, the maximum number of tasks activations may only increase or remain unchanged.

ii. x_i - The time intervals x_i which have to be investigated in order to find the worst-case mode change scenario are fixed values which remain unchanged during iterations.

iii. ϕ_i - The offsets ϕ_i for each added tasks are statically defined in the system model and remain unchanged during analysis.

iv. Because ϕ_i and x_i are constant and $\delta_i^-(q)$ may only decrease or remain unchanged during iterations the response time function (1) is order preserving only if the busy window function $w_i(q)$ is order preserving. See Step 2 below.

Step 2. The transition busy window $w_i(q)$, calculated with (4.15) which is:

$$w_i^{n+1}(q) = MW^i + BT_i(w_i^n(q)) + \sum_{\forall \tau_F \in hpl_F(i)} \eta_{\tau_F}^+(x_i) \cdot C_{\tau_F} + \\ \sum_{\forall \tau_U \in hpl_U(i)} \eta_{\tau_U}^+(w_i^n(q)) \cdot C_{\tau_U} + \\ \sum_{\forall \tau_A \in hpl_A(i)} \eta_{\tau_A}^+(w_i^n(q) - x_i - \phi_{\tau_A})_0 \cdot C_{\tau_A} \quad (2)$$

is order preserving if all its elements (i.e. the individual terms MW^i, $BT_i(w_i^n(q))$ and the three sum factors, are order preserving with respect to the analysis states.

i. The first term MW^i, given by

$$MW^i = \begin{cases} q \cdot C_i; & \text{if } (i == U) \mid\mid (i == F) \\ \min(q, \eta_i^+(w_i^n(q) - x_i - \phi_i)_0) \cdot C_i; & \text{if } (i == A) \end{cases}$$

captures the execution of the analyzed task during the investigated time interval and is composed of the constant factor C_i and the number of considered task activations q which can only increase or remain unchanged.

ii. The second term in (2), i.e. the blocking time $BT_i(w_i^n(q))$ of the analyzed task τ_i, corresponds to (4.26). Each blocking term of (4.26) is a function of:
   a. the load $\tilde{\eta}_j^+(w_i(q))$ imposed by other tasks τ_j in the system on the shared resources and of
   b. other parameters, which are
      - either constant during iterations, such as the parameters x_i or ϕ_i, the size of the critical sections ω_j^{LR}, ω_j^{GR} or the number of shared resource accesses per task instance n_i^G,
      - or order preserving, such that the number of considered task activations q

   Thus, the blocking time analysis equation $BT_i(w_i^n(q))$ is order preserving only if the shared resource request bound function $\tilde{\eta}_j^+$ (see a. above) is order preserving. This however, is inherent to (3.8) where an specific event model estimate η^+ is scaled by a constant factor or (3.9) where the number of issued shared resource requests increases with the size of the investigated time window, which is always divided to the constant factor d_{srr}.

iii. The third, fourth and fifth terms are sums, over the higher priority tasks mapped on the same resource as τ_i, which consider the order preserving function η^+ and the constant factors C, x_i and ϕ_i depending on the task types.

As all individual factors on the right hand side of (2) are order preserving and the addition and multiplication operators are also order preserving, the busy window analysis function (2) is order preserving. This proofs point iv. under Step 1.

From Step 1 and Step 2 all functions of the local response time analysis procedure are order preserving and all their input parameters form a complete partial order set. Theorem 4.10 follows. □

Theorem 4.10 proves that the two conditions of Corollary 2.2 are fulfilled for all components of the system-level analysis procedure (i.e. for the local analysis functions) and therewith for the global analysis function itself (according to Corollary 2.1).

Given the order-preservingness of the extended system-level analysis procedure the analysis will either converge towards a fixed point (i.e. all task activating event models η^+ and all shared resource request bounds $\tilde{\eta}^+$ have not changed after an iteration and lead to identical response-time analysis results), which represent a conservative solution, or the event model estimates grow to infinity, in which case the analysis will be stopped as soon as a real-time constraint (e.g. deadline of a task) is violated.

4.5.4 Experiments

To demonstrate the applicability and the benefits of the proposed approach we compare it to the currently available design procedure for AUTOSAR multi-core systems. The current design practice, which is not multi-mode aware, can safely handle the system in Figure 4.15 only by assuming that all tasks are always running on the two cores, i.e. by

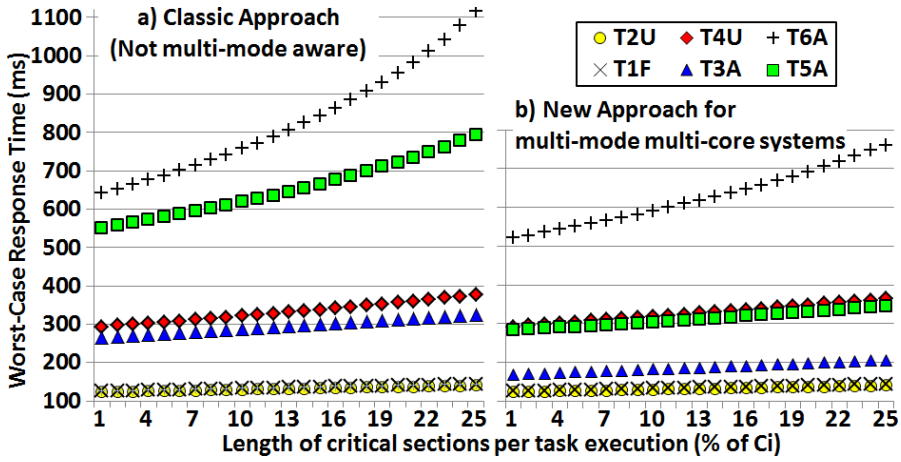


Figure 4.19: WCRTs of tasks depending on the critical sections length: a) current design practice; b) our approach for multi-mode multi-core systems.

modeling all tasks as unchanged, not only in the individual modes but also during the transition phase.

Hence, for the transition phase of the system in Figure 4.15, we apply both, (a) the classic response-time analysis method for the case where all tasks are modelled as unchanged and (b) our approach (in Section 4.5.2 and 4.5.3) which is able to handle the multi-mode behavior of the multi-core system.

For the evaluation we randomly generated test cases until we got 1000 schedulable configurations of the system in Figure 4.15. The test cases were generated such that: the load on each core was 50%; the load on a core was randomly distributed among tasks; the tasks' periods P_i were generated randomly between 100 and 1000ms; the tasks' execution times C_i were computed based on the tasks' periods and loads. Each task was randomly assigned an input jitter from the interval $[0, 2 \cdot P_i]$, i.e. we generated a burst of maximum 3 activations. Each task performs two requests for each LR and GR it uses during C_i. The total length of the critical sections per C_i was equally split among the number of requests. Based on the number and on the size of the critical sections the distance between every two requests d_{srr} was modelled such that critical sections are equally spread across the C_i. Thus, the load imposed on the shared resources was calculated with $\tilde{\eta}_i^+(\Delta t) = \lceil \Delta t / d_{srr} \rceil$.

For each test case, the total length of the task's critical sections was varied from 1% to 25% of the C_i. Figure 4.19 a) and b) depict the tasks' worst-case response times depending on the critical sections' length. For each task the average worst-case response time over the 1000 setups per critical section length is given.

As expected, independent of the design approach, increasing the size of the critical sections led to increased blocking times and therewith to increased response-times. However, when comparing the results of the two approaches, one can see that our proposed approach greatly takes advantage of its ability of handling the different types of tasks

across mode changes. Whereas for the higher priority tasks τ_{1F} and τ_{2U}, there is no difference, as the AUTOSAR spinlock-based arbitration always favours them, for the other tasks, the response-times computed with our approach are in average 30.5% lower. More exactly, there is an average improvement of 1.5% for task τ_{4U}, of 42% for task τ_{3A}, 53% for task τ_{5A} and 25.5% for task τ_{6A}.

4.6 Summary

This chapter addressed the timing behavior of multi-mode real-time systems in two parts.

The first part focused on the timing behavior of multi-mode distributed applications under asynchronous mode change protocols. In order to validate the timing behavior of such systems, the calculation of the mode change transition latencies is required in addition to schedulability analysis. Consequently, a solution was proposed for analyzing the duration of individual transitions phases between any two operational modes of a distributed system in which multi-mode applications consist of communicating tasks. In order to capture the dynamic effect of mode changes, the proposed solution relies on the compositional system-level analysis approach in [121, 64] and on the busy-window approach used for the analysis of each component in the system. Thus, the maximum busy-windows of tasks on a processor, calculated for the transition phases, were proven to upper-bound the settling time of a mode change on that processor. However, whereas the maximum busy-windows represent a conservative bound of the processor local transition latencies, these don't provide any information about the moment when the local transition latencies occur and how are these correlated with the local transition latencies of other resources in the system. Therefore, the local resource-level timing view was integrated into a global system-level timing view. By using a timing dependency graph, which indicates the functional and non-functional dependencies between the tasks in the system, and an algorithm, which considers these dependencies and computes the largest sum of local transition latencies along the paths of the graph, the local transition latencies of the tasks in the system are correlated. The largest sum obtained with the proposed algorithm was shown to upper-bound the duration of the mode change transition phase of the entire distributed system. Experimental results and the investigation of an automotive specific case study show the applicability of the proposed solution.

The second part of this chapter focused on the timing behavior of multi-mode applications mapped on multi-core systems with shared resources, a combination which was not considered so far in the research. The key challenge for providing safe timing guarantees for such setups is to jointly handle (i) the multi-core scheduling, (ii) the shared resource arbitration and (iii) the mode management. Specifications of the AUTOSAR standard introduced individual guidelines on all these three aspects, however, without to consider their inevitable interdependence in multi-mode multi-core systems. This chapter combined these elements and proposed an approach for safely handling inter-core and intra-core shared resources across asynchronous mode changes in multi-core systems. A corresponding solution for deriving blocking-times and response-times was also contributed.

In order to tackle the contention of tasks on the processor cores and on the shared resources, the blocking-time and response-time analysis equations were integrated in the iterative analysis procedure of the compositional system-level performance analysis methodology discussed in Chapter 2. Essentially, this timing analysis solution combines elements of the individual analysis approaches for (i) multi-core systems with shared resources in Chapter 3 and (ii) multi-mode systems in Section 4.4. The combination was made possible by the busy-window approach and the system-level analysis procedure on which the individual solutions are based. Section 4.5.3.4 showed that all analysis elements comply with the conditions of the fixed-point theory regarding the convergence of the iterative analysis procedures, fact that enables the calculation of conservative (i.e. safe) analysis results. The experimental part demonstrates the applicability of the proposed solution and its benefits against the current practice.

5 Conclusion

This thesis addresses the topic of performance analysis for static and multi-mode multi-core systems with shared resources such as implemented in modern automobiles. The steadily increasing number and complexity of functions implemented in various application domains, including the automotive domain, challenge the performance limitations of single-core processor devices and have already triggered a paradigm shift of the embedded system design towards multi-core architectures. However, while multi-core solutions are expected to deliver additional performance, their applicability in static and multi-mode real-time systems is questioned by the execution delay caused by the contention of software applications on shared multi-core components such as shared memories, I/O devices, coprocessors or semaphores. In this context, the development process of multi-core real-time systems asks for a careful investigation of their timing behavior. This requires appropriate solutions for timing and performance verification.

Previous work from academia and industry showed that formal performance analysis approaches are well suited for the analysis of distributed and multiprocessor real-time systems. The applicability of existing solutions is, however, limited as many system details are not covered on the modeling and analysis side. In this general context, this thesis contributes new analysis methods which extend the scope of formal performance analysis and enable the investigation of new design options for multi-core real-time systems, especially for those that adhere to the automotive AUTOSAR standard specifications.

The contributions of this thesis to the state of the art in the field of formal performance analysis are summarized in the following.

• In Chapter 3 novel approaches were proposed for the analysis of worst-case blocking-times and response-times of static real-time applications that share resources in partitioned multi-core systems. For this purpose a compositional performance analysis methodology was adopted and extended to take into account the contention of tasks on the processor cores and on the shared resources. The solutions presented in this thesis consider realistic applications models with tasks that exhibit arbitrary activations and deadlines, and rely on an enhanced model to capture the load imposed on shared units. The new methods support different combinations of processor scheduling policies and shared resource arbitration strategies, proposed by academia and industry.

Highly relevant is the compatibility of the proposed analysis methods with the specifications of the AUTOSAR standard, which defines the combination of preemptive, non-preemptive and cooperative core local scheduling with lock-based arbitration of core local shared resources and spinlock-based arbitration of inter-core shared resources. The applicability and usefulness of the contributed analysis solutions are highlighted by the experimental evaluation.

• Chapter 4 addressed the timing behavior of multi-mode systems in two steps.

Section 4.4 focused on the timing behavior of multi-mode distributed applications under asynchronous mode change protocols. For such systems, the settling time of a mode change, called mode change transition latency, is an important system parameter that was neglected before. However, in order to validate the timing behavior of such systems, the calculation of the mode change transition latencies is required in addition to schedulability analysis. This thesis proposed the first solution for analyzing the duration of individual transitions phases between any two operational modes of a distributed system in which multi-mode applications consist of communicating tasks. In order to capture the dynamic effect of mode changes, the proposed solution uses (i) a timing dependency graph, which indicates the functional and non-functional dependencies between the tasks in the system, and (ii) an algorithm, which considers these dependencies and sums up the worst-case timing behavior along the paths of the graph. In order to derive the worst-case timing behavior of individual tasks, the proposed solution relies on an existing compositional system-level analysis approach and on the busy-window approach used for the analysis of individual components in the system.

Experimental results and the investigation of an automotive specific case study exemplify the applicability of the mode change transition latency analysis for the design of automotive applications with engine-synchronous tasks.

Section 4.5 focused on the timing behavior of multi-mode applications mapped on multi-core systems with shared resources, a combination which was not considered so far in the research. The key challenge for providing safe timing guarantees for such setups is to jointly handle (i) the multi-core scheduling, (ii) the shared resource arbitration and (iii) the mode management. Specifications of the AUTOSAR standard introduced individual guidelines on all these three aspects, however, without to consider their inevitable interdependence in multi-mode multi-core systems. This chapter combined these elements and proposed an approach for safely handling inter-core and intra-core shared resources across asynchronous mode changes in multi-core systems. A corresponding solution for deriving blocking-times and response-times was also provided. The proposed timing analysis solution combines elements of the individual analysis approaches for multi-core systems with shared resources in Chapter 3 and for multi-mode systems in Section 4.4. This combination was enabled by the busy-window analysis approach and the compositional system-level analysis procedure on which the individual solutions are based.

The experimental part demonstrates the applicability of the proposed solution and its benefits against the current automotive practice.

• Relevant for the practical use of any performance analysis methods is an appropriate tool support. In the context of this thesis, the academic version of the SymTA/S tool, originally developed at TU Braunschweig, was adopted and extended with new modeling and analysis elements, which correspond to the theoretical research presented in

Chapters 3 and 4. Together with a test-case generator, implemented and connected with the SymTA/S tool, the performance analysis framework was used for the experimental evaluations presented in this thesis and in the publications underlying it.

To sum up, the contribution of this thesis is a comprehensive and flexible performance analysis framework for static and multi-mode real-time applications which share resources on multi-core systems. To enable the practical applicability, this framework and its components were primarily developed to suit the current practice in the automotive industry, particularily for the present multi-core architectures and AUTOSAR specifications. Furthermore, this framework can serve as an enabler for the introduction of new technologies and standards. Its flexibility permits the investigation of different design options and thus can be very helpful in defining ultimate directives for the industrial practice.

5.1 Future directions

Even if this thesis provides significant extensions of the scope of formal performance analysis methods, there are clearly aspects that were not considered and are of interest for further research activities.

For example, current specifications of the AUTOSAR standard mandates the implementation of spinlocks for inter-core synchronization, but doesn't specify details on the execution order of critical sections in case of conflicting accesses. However, the order of granting the locks is one essential design decision without which the prediction of the timing behavior is not possible. For the purpose of this thesis spinlocks were assumed assigned based on tasks priorities, assumption which maintains the compatibility with the state-of-the art priority based scheduling in the automotive design. The proposed analysis framework can be extended to consider other design options regarding the arbitration of spinlocks, an investigation of their benefits and drawbacks could help, if desired, to standardize the AUTOSAR spinlocks semantic.

Furthermore, the analysis of the mode change transition latencies presented in Section 4.4 is dedicated to multi-mode distributed applications without cyclic dependencies. A method that can analyze accurately systems comprising multi-mode distributed applications which contain cyclic dependencies would further extend the capabilities of formal performance analyses.

6 List of publications

This chapter lists publications of the author, first with relation to this thesis, then those without. Publications are ordered by date of appearance.

6.1 With Relation to Thesis

[1] Mircea Negrean, Sebastian Klawitter and Rolf Ernst, "Timing Analysis of Multi-Mode Applications on AUTOSAR conform Multi-Core Systems" in Proceedings of Design, Automation and Test in Europe (DATE), March 2013.

This paper introduces an approach for safely handling shared resources across asynchronous mode changes in AUTOSAR conform multi-core processors and a corresponding timing analysis solution. The contribution of this paper builds on the individual analysis solutions for multi-core systems in [2,7,10,11] and multi-mode systems in [4,5]. Its findings are elaborated in Section 4.5 in Chapter 4.

[2] Mircea Negrean and Rolf Ernst, "Response-Time Analysis for Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources" in Proc. of 7th IEEE International Symposium on Industrial Embedded Systems (SIES), (Karlsruhe, Germany), June 2012.

This paper contributes a timing analysis solution for partitioned multi-core systems with shared resources scheduled according to the static-priority non-preemptive scheduling. Its findings have been incorporated in Chapter 3, especially in Section 3.8. Togheter with the contribution of [11] and [12] below, the contribution of this paper paved the way for the timing analysis solution in Section 3.9 that covers the combination of preemptive and non-preemptive scheduling of next generation AUTOSAR conform automotive multi-core ECUs.

[3] Jonas Rox, Mircea Negrean, Simon Schliecker, and Rolf Ernst, "System level performance analysis for real-time multi-core and network architectures" in Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years), pp. 171-189, 2012.

This book chapter highlights extensions of the system level formal performance analysis approach SymTA/S, extensions that cover the analysis of multi-core architectures and the incorporation of modern communication stacks into system analysis. The content of this book chapter related to the multi-core topic builds on the contribution of the papers [7, 9, 10, 11, 12, 17] below.

[4] Mircea Negrean, Rolf Ernst and Simon Schliecker, "Mastering Timing Challenges for the Design of Multi-Mode Applications on Multi-Core Real-Time Embedded Systems" in 6th International Congress on Embedded Real-Time Software and Systems (ERTS), (Toulouse, France), February 2012.

This paper identifies similarities between the problem of scheduling automotive specific real-time applications which accommodate tasks with angular recurrence (i.e. engine-synchronous tasks implemented e.g. in automotive powertrain controllers) and the problem of scheduling multi-mode applications. An automotive specific case study explains and exemplifies how the formal analysis method in [5] can be applied for the design and analysis of multi-core real-time systems. The impact of shared resource is not considered here. The case study is incorporated in Section 4.4.4 in Chapter 4.

[5] Mircea Negrean, Moritz Neukirchner, Steffen Stein, Simon Schliecker and Rolf Ernst, "Bounding Mode Change Transition Latencies for Multi-Mode Real-Time Distributed Applications" in 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11), (Toulouse, France), September 2011.

This paper provides the first analysis method to bound the transition latency of asynchronous mode changes in distributed systems with communicating tasks. Its contribution was integrated in Chapter 4 of this thesis, especially in Section 4.4.

[6] Philip Axer, Jonas Diemer, Mircea Negrean, Maurice Sebastian, Simon Schliecker and Rolf Ernst, "Mastering MPSoCs for Mixed-Critical Applications", IPSJ Transactions on System LSI Design Methodology, vol. 4, pp. 91-116, August 2011.

This paper presents challenges and potential solutions of mixed-critical MPSoC designs. Especially, concerns of MPSoC architectural implications such the impact of shared resource contention on timing, NoC-based communication, multiple modes of operation and safety constraints are raised. The contribution related to the formal analysis of multi-core and multi-mode systems are related to Chapter 3 and 4.

[7] Mircea Negrean, Simon Schliecker and Rolf Ernst, "Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems", SAE International Journal of Passenger Cars - Electronic and Electrical Systems, vol.3, No.1, pp. 27-40, August 2010.

By using the modelling and analysis framework for multi-core systems with shared resources developed in previous own work, this paper investigates the impact of different design decisions regarding task scheduling and shared resource arbitration on the timing behavior of multi-core applications. The contribution of this paper is related to Chapter 3.

[8] Mircea Negrean, Simon Schliecker and Rolf Ernst, "Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems" in SAE 2010 World Congress and Exhibition Technical Papers, (Detroit, MI, USA), April 2010.

This paper was later accepted as SAE journal paper - see [7] above.

[9] Simon Schliecker, Mircea Negrean and Rolf Ernst, "Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems" in Proc. of Design, Automation, and Test in Europe (DATE), (Dresden, Germany), March 2010.

This paper contributes a formal method that captures more accurately the load imposed on shared resources in multi-core systems and thus enable for improved blocking time and response time analysis results. Key aspects of the improved shared resource load derivation discussed in Section 3.6 are exploited by the analysis methods presented in Chapter 3 and Chapter 4.

[10] Simon Schliecker, Mircea Negrean and Rolf Ernst, "Response Time Analysis in Multicore ECUs with Shared Resources", IEEE Transactions on Industrial Informatics, vol. 5, No. 4, November 2009.

This paper adresses the contribution of the conference paper [12] in the context of common automotive ECUs and formally reasons about the fixed-point solution on which the system-level analysis for multi-core systems with shared resource relies. Chapter 2 and Chapter 3 of this thesis elaborate further on the contribution of this paper.

[11] Simon Schliecker, Jonas Rox, Mircea Negrean, Kai Richter, Marek Jersak and Rolf Ernst, "System Level Performance Analysis for Real-Time Automotive Multi-Core and Network Architectures", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, No. 7, pp. 979-992, July 2009.

This paper highlights key challenges for the application of performance analysis in the automotive system design, identifies the need for well defined system timing models and discusses modelling and analysis extensions for networked architectures and multi-core systems with shared resources. These details are considered by the modelling and analysis approaches contributed across this thesis for today's and future automotive multi-core systems.

[12] Mircea Negrean, Simon Schliecker and Rolf Ernst, "Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources" in Proc. of Design, Automation, and Test in Europe (DATE), (Nice, France), April 2009.

This paper presents an approach to bound blocking times and response times of arbitrarily activated tasks in hard-real time multi-core systems with shared resources under partitioned static-priority preemptive scheduling and MPCP shared resource arbitration. Its contribution have been incorporated in Chapter 3, especially in Section 3.7.

6.2 Others

[13] Sophie Quinton, Torsten T. Bone, Julien Hennig, Moritz Neukirchner, Mircea Negrean und Rolf Ernst, "Typical Worst Case Response-Time Analysis and its Use in Automotive Network Design" in Design Automation Conference (DAC), 2014.

This paper applies typical worst-case analysis to investigate the effect of complex load patterns on the timing behavior of automotive CAN buses and shows how the necessary parameters can be derived and verified from traces and specifications.

[14] Sophie Quinton, Mircea Negrean and Rolf Ernst, "Formal Analysis of Sporadic Bursts in Real-Time Systems" in Proc. of Design, Automation and Test in Europe (DATE), March 2013.

This paper proposes a new method for the analysis of typical-case response-times in uni-processor real-time systems where task activation patterns may contain sporadic bursts. The method provides smaller typical-case response-times than the traditional worst-case analyses while the error remains at an acceptable level.

[15] Moritz Neukirchner, Mircea Negrean, Rolf Ernst and Torsten Bone, "Response-Time Analysis of the FlexRay Dynamic Segment ander Consideration of Slot-Multiplexing" in Proc. of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES), (Karlsruhe, Germany), June 2012, BEST PAPER AWARD.

This paper presents a response-time analysis approach for the dynamic segment of the automotive communication network FlexRay and shows its applicability with a realistic automotive case-study provided by Daimler AG.

[16] Jonas Diemer, Jonas Rox, Mircea Negrean, Steffen Stein and Rolf Ernst, "Real-Time Communication Analysis for Networks with Two-Stage Arbitration" in Proc. of the 9th ACM International Conference on Embedded Software (EMSOFT), (Taipei, Taiwan), pp. 243-252, ACM, October 2011, ISBN 978-1-4503-0714-7.

This paper introduces a timing analysis method for networks with multi-stage arbitration mechanisms. The proposed solution maps the multi-stage arbitration analysis to a schedulability analysis of multiprocessors with shared resources.

[17] Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin and Rolf Ernst, "Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip" in Proc. of the 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS), (Atlanta, GA), October 2008.

In this paper a formal performance analysis is applied to a realistic embedded multiprocessor system-on-chip with shared resources. Benchmark results show that corner case coverage of the considered formal analysis method is supplied with a very high accuracy, allowing to quickly investigate architectural alternatives.

List of Figures

1.1 Growing complexity of E/E components and network communication (Source: Daimler AG Group Research and Advanced Engineering [155]) 11
1.2 Top 10 above average automotive applications growth rates [163] 12
1.3 Multi-core systems - use cases 14
1.4 Block diagram Infineon Aurix multi-core architecture (Source [66]) 15
1.5 a) Tasks statically mapped on different cores share a common resource SR; b) Single-core vs. multi-core execution: Conflicting accesses for inter-core shared resources delay the completion of higher priority tasks. 17
1.6 Task mapping in individual modes and during the transition between them. 19
1.7 Timing aspects in the system development process according to the V-Model. 21

2.1 Event stream representation. 35
2.2 Example of task execution and the associated upper event arrival function η^+ and shared resource request bound function $\tilde{\eta}^+$. 36
2.3 Example of a dual-core processor with tasks which access local (LR) and global shared resources (GR). 37
2.4 Example of a task execution and corresponding extended task state model (OSEK state model [100]). 38
2.5 a) Classic CPA procedure; b) Extended CPA procedure for multi-core systems with shared resources. 40

3.1 a) Example system with three single-core CPUs and one multi-core CPU connected to a communication bus. b) Detailed view of the multi-core CPU with tasks accessing local and global shared resources. 59
3.2 Granting resources in a) FCFS manner and b) priority-based manner when tasks on different cores attempt to lock the same global shared resource. 63
3.3 a) Deadlock due to waiting for an unreleased global shared resource. b) Using priority ceilings avoids unbounded priority inversion and deadlock situations. 64
3.4 Blocking due to global shared resources when a) suspending and b) spinning (busy-waiting). 65
3.5 a) Preemption of lower priority tasks during busy-wait execution. b) Preemption of higher priority tasks during busy-wait execution when lower priority tasks receive the requested resource. 66

3.6 a) Preemption of a critical section by other critical section with higher priority. b) Forbid preemption of critical sections. c) Preemption of normal execution by a critical section. 67
3.7 Deadlock situations when nesting a) global and b) local shared resources. 69
3.8 Scheduling example and maximum busy windows for a task τ_i on a single-core processor scheduled according to a) SPP and b) SPNP scheduling. . 70
3.9 Scheduling example and maximum busy windows for a task τ_i on a single-core processor under SPP scheduling and IPCP shared resource arbitration. 71
3.10 Load imposed by task τ_1 on the shared resource GR1. 75
3.11 Example: minimum and maximum possible distance between two requests for the global resource *GR1* within the core execution time C_1. 77
3.12 Conflicting accesses from tasks mapped on different cores. 84
3.13 Critical instant and busy window for a task τ_i in a partitioned multi-core system with cores individually scheduled according to the SPNP scheduling. 89
3.14 Example of a task instance with two equally long runnables, each performing two requests for GRs. 92
3.15 Dual-core ECU with tasks accessing local and global shared resources. . . 94
3.16 Scheduling example on Core 1 where tasks τ_4 and τ_6 a) are fully non-preemptive and b) are cooperative to each other. 95
3.17 Critical instant example for task τ_6 in the multi-core system in Figure 3.15.104
3.18 Dependencies in the response-time analysis procedure. 111
3.19 Benefit of using the minimum distance between requests d_{srr} in the shared resource request derivation on the tasks' worst-case response times: - "classic" - response times obtained with the analysis in [116] - "improved" - response times obtained with the new analysis in Section 3.7. 116
3.20 Worst-case response time depending on the critical sections length for the tasks in the system Figure 3.1b) under partitioned SPP scheduling and MPCP shared resource arbitration. 118
3.21 Worst-case response time depending on the critical sections length for the tasks in the system Figure 3.1b) under partitioned SPNP scheduling and MLP-NP shared resource arbitration. 119
3.22 a) Worst-case response time of the individual tasks and b) utilization of the individual cores depending on the critical sections length. 120
3.23 Multi-core ECU with tasks accessing local and global shared resources. . . 121
3.24 WCRTs under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) scheduling for randomly generated parameter for the dual-core (DC) and multi-core (MC) setups in Fig 3.15 and Figure 3.23. 124
3.25 WCRTs depending on the critical sections length in the dual-core (DC) and multi-core (MC) setups under fully preemptive (FP), cooperative (Coop), mixed-preemptive (MP) and fully non-preemptive (FNP) AUTOSAR scheduling. (*Note the difference between the scale range in case of DC and MC analysis results*). 127

4.1 Distributed system performing a transition between two modes M1 and M2. During the transition phase tasks of both modes execute on the system. 136
4.2 a) Illustration of a possible settling behavior for tasks τ_{4U} and τ_{7U}. b) Potential mode change time line for τ_{4U} and τ_{7U} in the context of the system transition latency. 139
4.3 Scheduling example during a mode change where MCR_i coincides with the 3^{rd} activation of the finished task - **Worst-case mode change scenario**. 143
4.4 Scheduling example during a mode change where MCR_i coincides with the 2^{nd} activation of the finished task. 143
4.5 Scheduling example during a mode change where MCR_i occurs later than the 3^{rd} activation of the finished task. 143
4.6 Timing dependency graph for the system example in Figure 4.1. 148
4.7 Task transition latencies in the context of the system transition phase. . . 151
4.8 Mode change system transition latencies depending on the activation backlog of the finished task τ_{1F}. 153
4.9 Mode change system transition latencies depending on the activation backlog of task τ_{1F} modelled as unchanged task. 154
4.10 Illustration of a dual-core processor with inter-core communication. 154
4.11 a) Time intervals between two constant engine-speed values. b) Example of engine-speed variation over time during an acceleration phase. 155
4.12 Workload to be processed on each core. Increasing engine speed leads to higher rate of activation for engine-synchronous tasks. In addition task modes lead to varied workload. Critical load on Core 1 is reached around 3500 and 5500 rpm. 156
4.13 Multiple mode changes in order to avoid overload at different RPM values. 159
4.14 a) Mode changes shall be initiated at X rpm during acceleration and at Y rpm during deceleration in order to avoid a non-schedulable situation at CP rpm. b) Complex mode changes are possible if there is enough headroom for mode change transition latencies. 160
4.15 Multi-mode multi-core system during a transition phase. 162
4.16 Scheduling examples for the dual-core system in Figure 4.15 when a) task priorities correspond to the system model in Figure 4.15; b) task τ_{3A} has higher priority than τ_{1F}, i.e. their priorities are interchanged, and the offset of the added task remains unchanged, i.e. $\Phi_{3A} = \Phi_{1A}$; and c) priorities of tasks τ_{3A} and τ_{1F} are interchanged and the offset of the added task is larger in comparison to case b), i.e. $\Phi'_{1A} > \Phi_{1A}$. 164
4.17 Scheduling example for the case in the system in Figure 4.15 there would be a third core on which a lower priority added task τ_{7A} would be started during the transition phase. 165
4.18 Scheduling example for a task τ_i during a mode change where MCR coincides with the 2nd activation of the higher priority finished task. 168
4.19 WCRTs of tasks depending on the critical sections length: a) current design practice; b) our approach for multi-mode multi-core systems. . . . 180

List of Tables

3.1 Parameters of the Multi-Core System Model 61
3.2 Particular configuration of the parameters for the system in Figure 3.1b under partitioned SPP scheduling and MPCP shared resource arbitration. 115
3.3 Accesses to the shared resources for the task in Figure 3.1b. 117
3.4 Particular configuration of the parameters for the system in Figure 3.1b under partitioned SPNP scheduling and MLP-NP shared resource arbitration. 120
3.5 Particular configuration for the systems in Figure 3.15 and Figure 3.23 . . 122

4.1 Parameters for the system in Figure 4.1 152
4.2 Analysis results: Task and system transition latencies 153
4.3 Parameters of engine synchronous tasks 155

Bibliography

[1] ARINC 653: Avionics Application Software Standard Interface. http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf (retrieved 28.03.2013).

[2] AbsInt. aiT WCET Analyser. http://www.absint.com/ait/ (retrieved 28.03.2013).

[3] K. Albers, F. Bodmann, and F. Slomka. Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–106, Dresden, Germany, July 2006.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193–202, Dec. 2001.

[5] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 337–346, 12-14 Dec. 2000.

[6] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50 In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 33–40, 2-4 July 2003.

[7] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *21st Intl. Conference on VLSI Design*, Hyderabad, India, January 2008.

[8] AUTOSAR. AUTomotive Open System ARchitecture. http://www.autosar.org/ (retrieved 04.03.2013).

[9] AUTOSAR. Specification of Timing Extensions V1.2.0 R4.0 Rev 3. http://www.autosar.org/ (retrieved 28.03.2013).

[10] AUTOSAR. Guide to Modemanagement R4.0 v1.0.0. http://www.autosar.org/, October 2011.

[11] AUTOSAR. Requirements on Operating System R4.0 v3.0.0. http://www.autosar.org/, October 2011.

[12] AUTOSAR. Specification of Operating System R4.0 v5.0.0. http://www.autosar.org/, November 2011.

[13] AUTOSAR. Specification of RTE R4.0 v3.2.0. http://www.autosar.org/, November 2011.

[14] C. Baier and JP. Katoen. *Principles of Model Checking*. Publisher, 2008.

[15] Sanjoy K. Baruah. The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Syst.*, 32(1-2):9–20, 2006.

[16] Sanjoy K. Baruah and Nathan Wayne Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems*, 36(3):199–226, 2007.

[17] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. *Proceedings 8th International Workshop Software and Compilers for Embedded Systems (SCOPES), LNCS 3199*, chapter 6: Predictable Embedded Multiprocessor System Design, pages 77–91. Springer, Amsterdam, The Netherlands, September 2004.

[18] Marko Bertogna and Michele Cirinei. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 149–160, Tucson, Arizona, USA, December 2007.

[19] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, May 2005.

[20] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56, Daegu, Korea, August 2007.

[21] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

[22] B. Brandenburg. Improved Analysis and Evaluation of Real-Time Semaphore Protocols for P-FP Scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013.

[23] B. Brandenburg and J. H. Anderson. A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 105–124, Luxor, Egypt, December 2008. Springer-Verlag.

[24] B. Brandenburg and James. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, pages 1–66, 2012.

[25] B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 342–353, St. Louis, MO, USA, April 2008.

[26] B. Brandenburg, John M. Calandrino, and James H. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Real-Time Systems Symposium (RTSS)*, pages 157–169, Barcelona, Spain, Nov. 30 - Dec. 3, 2008.

[27] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(12):1 – 19, 2008.

[28] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356 –373, feb. 2007.

[29] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29(1):5–26, January 2005.

[30] J.M. Calandrino, J.H. Anderson, and D.P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *19th Euromicro Conference on Real-Time Systems, (ECRTS)*, pages 247–258, 2007.

[31] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 30–1 – 31–19, 2004.

[32] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-based Embedded System Designs. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 190–195, 2003.

[33] Chia-Mei Chen and Satish K. Tripathi. Multiprocessor Priority Ceiling Based Protocols. Technical report, University of Marylands, 1994.

[34] Grimal F. Leydier T. Mader R. Wirrer G. Claraz, D. Introducing Multi-Core at Automotive Engine Systems. In *7th Int. Congress on ERTS*^{2}, February 2014.

[35] E. Coffman, J. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Combinatorial analysis, handbook of combinatorial optimization., 1998.

[36] R.L. Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[37] R.L. Cruz. A calculus for network delay. ii. network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[38] Andreas E. Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen, and Jaco van de Pol. Multi-core reachability for timed automata. In *Proceedings of the 10th international conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'12, pages 91–106, Berlin, Heidelberg, 2012. Springer-Verlag.

[39] A. David, J.I. Rasmussen, K.G. Larsen, and A. Skou. *Model-Based Design for Embedded Systems*, chapter Model-based Framework for Schedulability Analysis Using Uppaal 4.1. C R C Press LLC, 2009.

[40] Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40, January 2011.

[41] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[42] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller

Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007.

[43] U.M.C. Devi, H. Leontyev, and J.H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, Dresden, Germany, 2006. IEEE Computer Society Washington, DC, USA.

[44] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[45] Arvind Easwaran and Bjorn Andersson. Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386, Washington, DC, USA, 2009.

[46] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 90–99, 2010.

[47] FlexRay Consortium. FlexRay Communications System - Protocol Specication Version 2.1 Revision A. http://www.flexray.com/ (retrieved 28.03.2013), December 2005.

[48] Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling, November 1993.

[49] Freescale Semiconductors. Freescale Medical/Healthcare Applications. http://www.freescale.com (retrieved 04.03.2013), September 2011.

[50] Freescale Semiconductors. MPC5676R: Qorivva 32-bit MCU for Powertrain Applications. http://www.freescale.com (retrieved 04.03.2013), October 2011.

[51] Freescale Semiconductors. PXS30: Power Architecture Safety MCU, 180 MHz, Dual-Locking Core, 2MB On-Chip Flash. http://www.freescale.com (retrieved 04.03.2013), October 2011.

[52] Freescale Semiconductors. Rationale for Multicore Architectures in Automotive Apps. Freescale Technology Forum, http://www.freescale.com/files/training_pdf/WBNR_FTF11_AUT_F0166.pdf (retrieved 04.03.2013), June 2011.

[53] Freescale Semiconductors. MPC5676R Data Sheet: Advanced Information (Rev. 3). http://www.freescale.com (retrieved 04.03.2013), September 2012.

[54] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. *9th IEEE Real-Time and Applications Symposium (RTAS)*, pages 189–198, May 2003.

[55] Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed Model Checking with Abstractions: Towards Worst-Case Response Time Analysis in Resource-Sharing Manycore Systems. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 63–72, Tampere, Finland, Oct 2012.

ACM.

[56] M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, and J.M. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*, pages 125–134, 2001.

[57] Joël Goossens and Pascal Richard. Partitioned scheduling of multimode multiprocessor real-time systems with temporal isolation. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, RTNS '13, pages 297–305, New York, NY, USA, 2013. ACM.

[58] K. Gresser. An Event Model for Deadline Verification of Hard Real-Time Systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, 1993.

[59] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'07, pages 263–272, Berlin, Heidelberg, 2007. Springer-Verlag.

[60] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 387–397, Washington, DC, USA, 2009.

[61] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536 – 546, 2011.

[62] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New Schedulability Test Conditions for Non-preemptive Scheduling on Multiprocessor Platforms. In *29th IEEE Real-Time Systems Symposium (RTSS)*, pages 137–146, Washington, DC, USA, 2008.

[63] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 179–179, 2006.

[64] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proc. Comp. and Digital Tech.*, 152(2):148–166, Mar. 2005.

[65] Rafik Henia and Rolf Ernst. Scenario Aware Analysis for Complex Event Models and Distributed Systems. In *Proc. 28th IEEE RTSS*, Dec. 2007.

[66] Infineon Technologies. AURIX - Safety joins Performance. http://www.infineon.com/aurix (retrieved 04.03.2013), July 2012.

[67] Infineon Technologies. Highly Integrated and Performance Optimized - 32-bit Microcontrollers for Automotive and Industrial Applications. http://www.infineon.com (retrieved 04.03.2013), August 2012.

[68] M. Jersak. Compositional performance analysis for complex embedded applications. In *Dissertation 2004, Technische Universität Braunschweig*, 2004.

[69] Bengt Jonsson, Simon Perathoner, Lothar Thiele, and Wang Yi. Cyclic dependencies in modular performance analysis. In *Proc. of the 8th ACM International Conference on Embedded software (EMSOFT)*, pages 179–188, New York, NY, USA, October 2008. ACM.

[70] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[71] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[72] Hermann Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered Ethernet (TTE) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2005.*, pages 22–33, 2005.

[73] Hermann Kopetz and Gnther Bauer. The Time-Triggered Architecture. In *Proceedings of the IEEE*, pages 112–126, 2003.

[74] K. Lakshmanan, R. Rajkumar, and J.P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *21st Euromicro Conference on Real-Time Systems, (ECRTS)*, pages 239–248, 2009.

[75] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 469–478, 2009.

[76] S. Lauzac, R. Melhem, and D. Mossé. An Improved Rate-Monotonic Admission Control and Its Applications. *IEEE Transactions on Computers*, 52(3):337–350, March 2003.

[77] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer-Verlag, Berlin, Heidelberg, 2001.

[78] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Real-Time Systems Symposium (RTSS)*, pages 201–209, Lake Buena Vista, Florida, USA, Dec 1990.

[79] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[80] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Syst.*, 24(1):5–28, 2003.

[81] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):642–653, July 2004.

[82] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *31st IEEE Real-Time Systems Symposium (RTSS)*, pages 339–349, 2010.

[83] J. Becker M. Jersak K. Richter M. Khl M. Traub, V. Lauer. Using timing analysis

for evaluating communication behavior and network topologies in an early design phase of automotive electric/electronic architectures. *SAE World Congress, Detroit*, April 2009.

[84] G. Macariu and V. Cretu. Limited Blocking Resource Sharing for Global Multiprocessor Scheduling. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262–271, 2011.

[85] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Embedded Systems. Springer Verlag, ISBN: 978-94-007-0256-1, 2 edition, December 2011.

[86] Mobile World Congress. http://www.mobileworldcongress.com/.

[87] A. K. Mok. *Fundamental Design Problems Of Distributed Systems For The Hard Real-Time Environment*. PhD thesis, Cambridge, MA, USA, 1983.

[88] M. Negrean and R. Ernst. Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 191–200, 2012.

[89] M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *IEEE Conf. on ETFA*, pages 1–10, Sept. 2011.

[90] M. Negrean, S. Schliecker, and R. Ernst. Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Nice, France, April 2009.

[91] M. Negrean, S. Schliecker, and R. Ernst. Mastering Timing Challenges for the Design of Multi-Mode Applications on Multi-Core Real-Time Embedded Systems. In *6th Int. Congress on ERTS*^{2}, February 2012.

[92] Mircea Negrean, Sebastian Klawitter, and Rolf Ernst. Timing Analysis of Multi-Mode Applications on AUTOSAR conform Multi-Core Systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, March 2013.

[93] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 3(1):27–40, August 2010.

[94] V. Nelis, B. Andersson, J. Marinho, and S.M. Petters. Global-EDF Scheduling of Multimode Real-Time Systems Considering Mode Independent Tasks. In *Proc. of 23rd ECRTS*, pages 205 – 214, july 2011.

[95] V. Nelis, J. Goossens, and B. Andersson. Two Protocols for Scheduling Multi-mode Real-Time Systems upon Identical Multiprocessor Platforms. In *Proc. of 21st ECRTS*, pages 151–160, July 2009.

[96] F. Nemati, M. Behnam, and T. Nolte. Independently-Developed Real-Time Systems on Multi-cores with Shared Resources. In *23rd Euromicro Conference on*

Real-Time Syst. (ECRTS), pages 251 – 261, July 2011.

[97] Moritz Neukirchner, Mircea Negrean, Rolf Ernst, and Torsten Bone. Response-time analysis of the FlexRay dynamic segment under consideration of slot-multiplexing. In *Proc. of 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Karlsruhe, Germany, June 2012. BEST PAPER AWARD.

[98] Nvidia Tegra 4. http://www.nvidia.com/object/tegra-4-processor.html, March 2013.

[99] D.I. Oh and TP Bakker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, 1998.

[100] OSEK Consortium. OSEK OS Specification v2.2.3. http://www.osek-vdx.org/, February 2005.

[101] J.C. Palencia Gutierrez, J.J. Gutierrez Garcia, and M. Gonzalez Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, pages 136 –143, June 1997.

[102] PG Paulin, C. Pilkington, and E. Bensoudane. StepNP: a system-level exploration platform for network processors. *Design & Test of Computers, IEEE*, 19(6):17–26, 2002.

[103] P. Pedro. *Schedulability of Mode Changes In Flexible Real-Time Distributed Systems*. PhD thesis, University of York, Sep. 1999.

[104] P. Pedro and A. Burns. Schedulability Analysis for Mode Changes in Flexible Real-Time Systems. In *Proc. of the 10th Euromicro Workshop on Real-Time Systems*, pages 172 –179, June 1998.

[105] R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 741–746, 2010.

[106] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael Gonzalez Harbour. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1-2):27–49, 2009.

[107] Linh T. X. Phan, Insup Lee, and Oleg Sokolsky. A Semantic Framework for Mode Change Protocols. In *Proceedings of the 7th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 91–100, Washington, DC, USA, 2011. IEEE Computer Society.

[108] Linh T.X. Phan, Insup Lee, and Oleg Sokolsky. Compositional Analysis of Multi-Mode Systems. In *Proc. of 22nd ECRTS*, July 2010.

[109] P. Podevin, G. Descombes, P. Marez, and Dubois. F. A study of turbocharged diesel engine during sudden acceleration. set up and exploitation of a specific test rig. *in Internal Combustion Engine Division of ASME*, 1999.

[110] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the

synthesis of multi-cluster distributed embedded systems. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 184–189, 2003.

[111] T. Pop, P. Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *10th International Symposium on Hardware/Software Codesign (CODES)*, pages 187–192, 2002.

[112] Dev Pradhan. Multicore processors bring innovation to medical imaging. White Paper:Texas Instruments, May 2010.

[113] Qualcomm. Snapdragon s4. http://www.qualcomm.com/snapdragon/processors, March 2013.

[114] Razvan Racu, Li Li, Rafik Henia, Arne Hamann, and Rolf Ernst. Improved Response Time Analysis of Tasks Scheduled under Preemptive Round-Robin. In *International Conference on Hardware-Software Codesign and System Synthesis*, pages 179–184, Salzburg, Austria, October 2007.

[115] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[116] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publ. Norwell, MA, USA, 1991.

[117] R. Rajkumar, Lui Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, pages 259–269, 1988.

[118] Jorge Real and Alfons Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

[119] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *24th IEEE Real-Time Systems Symposium*, pages 236–245, Dec. 2003.

[120] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proc. of the 39th Conference on Design automation (DAC)*, pages 287–292. ACM New York, NY, USA, 2002.

[121] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technische Universität Braunschweig, 2004.

[122] Kai Richter, Marek Jersak, and Rolf Ernst. Learning Early-Stage Platform Dimensioning From Late-Stage Timing Verification. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Nice, France, April 2009.

[123] ROBERT N. CHARETTE. This car runs on code. IEEE Spectrum, Inside Technology, February 2009.

[124] Russell Fish. EDN: The future of computers - Part 1: Multicore and the Memory Wall. http://www.edn.com/design/systems-design/4368705/The-future-of-computers–Part-1-Multicore-and-the-Memory-Wall (retrieved 28.03.2013), November 2011.

[125] Sandia National Laboratories. More chip cores can mean slower supercomputing. https://share.sandia.gov/news/resources/news_releases/more-chip-cores-can-mean-slower-supercomputing-sandia-simulation-shows/ (retrieved 28.03.2013), January 2009.

[126] Alberto Sangiovanni-Vincentelli, Haibo Zeng, Marco Di Natale, and Peter Marwedel. *Embedded Systems Development - From Functional Methods to Implementations*. Springer, 2013. ISBN 978-1-4616-3878-6.

[127] Oliver Scheickl. *Timing Constraints in Distributed Development of Automotive Real-time Systems*. PhD thesis, Technische Universität München, 2011.

[128] Oliver Scheickl, Christoph Ainhauser, and Peter Gliwa. Tool Support for Seamless System Development based on AUTOSAR Timing Extensions. In *6th Int. Congress on ERTS*^{2}, February 2012.

[129] S. Schliecker, M. Ivers, and R. Ernst. Memory Access Patterns for the Analysis of MPSoCs. *Circuits and Systems, 2006 IEEE North-East Workshop on*, pages 249–252, 2006.

[130] S. Schliecker, M. Negrean, and R. Ernst. Response Time Analysis on Multicore ECUs with Shared Resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, November 2009.

[131] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System Level Performance Analysis for Real-Time Automotive Multicore and Network Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979–992, July 2009.

[132] Simon Schliecker. *Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources*. PhD thesis, Technische Universität Braunschweig, 2011.

[133] Simon Schliecker and Rolf Ernst. Real-Time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions on Embedded Computing Systems (Special Issue on Model Driven Embedded System Design)*, 10-2(22), December 2010.

[134] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems. In *Proc. of Design, Automation, and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[135] Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip. In *6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta, GA, October 2008.

[136] Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proc. 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta, GA, October 2008.

[137] A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In

Design Automation Conference (DAC), 2010 47th ACM/IEEE, pages 332–337, 2010.

[138] SGS_TÜV Saar GmbH. ISO26262 - Functional Safety Automotive. http://www.sgs-tuev-saar.com/ (retrieved 08.02.2014).

[139] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham. Mode Change Protocols for Priority-Driven Preemptive Scheduling. *Real-Time Systems*, 1:243–264, 1989.

[140] John A. Stankovic and K. Ramamritham, editors. *Tutorial: hard real-time systems.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[141] Jan Staschulat, Simon Schliecker, Matthias Ivers, and Rolf Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In *WCET*, 2005.

[142] Steffen Stein. *Allowing Flexibility in Critical Systems: The EPOC Framework.* PhD thesis, Technische Universität Braunschweig, 2012.

[143] Steffen Stein, Jonas Diemer, Matthias Ivers, Simon Schliecker, and Rolf Ernst. On the Convergence of the SymTA/S analysis. Technical report, Technische Universität Braunschweig, Germany, Nov. 2008.

[144] Steffen Stein, Moritz Neukirchner, Harald Schrom, and Rolf Ernst. Consistency Challenges in Self-Organizing Distributed Hard Real-Time Systems. *in Workshop on Self-Organizing Real-Time Systems (SORT)*, 2010.

[145] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling. In *Design, Automation Test in Europe (DATE)*, pages 99–104, April 2009.

[146] Xian-He Sun and Yong Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, February 2010.

[147] Symtavision GmbH. SymTA/S tool. http://www.symtavision.com/ (retrieved 28.03.2013).

[148] Alfred Tarski. A lattice-theoretical fixxpoint theorem and its applications. *Pacific J. Math.*, 5:285 – 309, 1955.

[149] The V-Model. http://www.v-modell.iabg.de/ (retrieved 28.03.2013).

[150] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[151] TIMMO-2-Use. TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases. http://www.timmo-2-use.org/ (retrieved 28.03.2013).

[152] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[153] K. W. Tindell, A. Burns, and A. J. Wellings. Mode Changes in Priority Preemptively Scheduled Systems. In *Proc. of the Real-Time Systems Symposium*, pages 100–109, 1992.

[154] K. W. Tindell, A. Burns, and A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[155] Torsten Bone. Applying Timing Analysis to Vehicle Networking at Daimler Group Research and Advanced Engineering. Symtavision News Conference 30.09.2010.

[156] A. Wieder and B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, December 2013.

[157] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):966–978, July 2009.

[158] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3):36:1–36:53, May 2008.

[159] Hang Yin, Etienne Borde, and Hans Hansson. Composable mode switch for component-based systems. In Linh TX Phan Sebastian Fischmeister, editor, *3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011)*, pages 19–22, April 2011.

[160] Hang Yin and Hans Hansson. Timing analysis for mode switch in component-based multi-mode systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS12)*, pages 255–264. IEEE Computer Society, July 2012.

[161] Patrick Meumeu Yomsi, Vincent Nélis, and Joël Goossens. Scheduling Multi-Mode Real-Time Systems upon Uniform Multiprocessor Platforms. *CoRR*, abs/1004.3687, 2010.

[162] Patrick Meumeu Yomsi, Vincent Nélis, and Joël Goossens. Scheduling multi-mode real-time systems upon uniform multiprocessor platforms. In *IEEE Conf. on ETFA*, pages 1–8, Sept. 2010.

[163] ZVEI – Zentralverband Elektrotechnik- und Elektronikindustrie e. V. Progress Report on the Application Group Automotive 2011/2012. http://www.zvei.org (retrieved 2013-02-28). May 2012.